jamk.fi

Tietoturva Kubernetes-konttiympäristössä

Eerik Kuoppala

Opinnäytetyö
Joulukuu 2017
Tekniikan ja liikenteen ala
Insinööri (AMK), Tietotekniikan tutkinto-ohjelma
Tietoverkkotekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Kuoppala, Eerik	Opinnäytetyö, AMK	Joulukuu 2017
	Sivumäärä 99	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: kyllä

Työn nimi
Tietoturva Kubernetes-konttiympäristössä

Tutkinto-ohjelma
Tietotekniikan (Tietoverkkotekniikan) koulutusohjelma

Työn ohjaaja(t)
Kokkonen, Tero; Rantonen, Mika

Toimeksiantaja(t)
Protacon Solutions Oy

Tiivistelmä

Opinnäytetyön toimeksiantajana oli Protacon Solutions Oy, joka tarjoaa ohjelmisto- ja ICT-palveluita.

Tavoitteena oli etsiä Kubernetekseen liittyviä käytännöllisiä tietoturva-asetuksia huomioiden eri kriteerejä ja työkalu havaitsemaan tietoturvahaavoittuvaisia Docker imageita. Tutkimuksissa testattiin erilaisia potentiaalisia tietoturva-asetuksia, joita voisi hyödyntää olemassa olevassa Googlen Kubernetes konttiympäristössä ja mahdollisesti tulevassa oman raudan ympäristössä. Lisäksi tehtiin laadullinen vertailu Docker imageiden haavoittuvuusskannereista, joka tarkistaa tietoturvahaavoittuvuudet vastarakennetusta Docker imagesta.

Opinnäytetyö toteutettiin etsimällä aineistoa Kuberneteksen tietoturvasta eri tietolähteistä ja testaamalla niiden toimivuutta erillisellä Kubernetes klusterilla. Haavoittuvuusskannerit asennettiin, testattiin ja sitten vertailtiin niiden eri ominaisuuksien perusteella. Niistä valitaan yksi, joka otetaan käyttöön osaksi CI/CD-prosessia.

Tuloksena oli tietoturvallisia tapoja suojata Kubernetes konttiympäristöä, mitä käytetään myös jatkossa. Lisäksi vertailusta otettu käyttöön Docker imageiden haavoittuvuusskanneri CI/CD-prosessissa, mikä toimii automaattisesti ja estää tietoturva-aukot konttiympäristöstä. Tietoturva-aukot saadaan selville haavoittuvuusskannerin luomasta raportista. Tietoturvakäytänteet ja käyttöohjeet dokumentoitiin Protaconin sisäiseen käyttöön.

Avainsanat (asiasanat)
Kubernetes, Kontti, Tietoturva, Docker, haavoittuvuusskannaus, Clair, Owasp

Muut tiedot

jamk.fi

Description

Author(s)	Type of publication	Date
Kuoppala, Eerik	Bachelor's thesis	December 2017
		Language of publication: Finnish
	Number of pages: 99	Permission for web publication: yes

Title of publication

Security in Kubernetes container environment

Degree programme

Information Technology

Supervisor(s)

Kokkonen, Tero; Rantonen, Mika

Assigned by

Protacon Solutions Oy

Abstract

The bachelor's thesis was assigned by Protacon Solutions Ltd that offers software and ICT services.

There were two objectives in the bachelor thesis. The first objective was to find practical security configurations to Kubernetes when considering different criteria. The second objective was to find tools to detect security vulnerabilities in Docker images. The purpose of the research was to test different potential security configurations which would be made use of in existing Google's Kubernetes container environment and maybe in upcoming on-premise solution. The qualitative comparison was made of Docker image vulnerability scanners, which check security flaws from built Docker images.

The material of Kubernetes security from different information sources was found and its functionality were tested in a separate Kubernetes cluster. The vulnerability scanners were installed, tested and compared based on their features. One of them was chosen to be part of the CI/CD process.

The study resulted in secured practices to protect Kubernetes container environment that will also be used in the future. The vulnerability scanner of Docker images was selected from the qualitative comparison and set in motion in CI/CD process. It works automatically and prevents the security holes from the container environment. Security vulnerabilities were found out from a report created by the vulnerability scanner. The security practices and operation manual were documented for Protacon's internal use.

Keywords/tags (subjects)

Kubernetes, Container, Security, Docker, Vulnerability scan, Clair, Owasp

Miscellaneous

Sisältö

Kuviot

Lyhenteet

API	Application Programmable Interface
BID	Bugtraq ID
CD	Continuous Delivery
CI	Continuous Integration
CIA	Confidentiality, Integrity and Availability
CIS	Center for Internet Security
CLI	Command Line Interface
CPE	Common Platform Enumeration
CVE	Common Vulnerabilities and Exposures
GKE	Google Container Engine
HTML	Hypertext Markup Language
REST API	Representational state transfer Application Programmable Interface
YAML	YAML Ain't Markup Language

1 Lähtökohdat ja toimeksianto

1.1 Työn kuvaus

Docker tarjoaa järjestelmällisen tavan automatisoida nopeampaa sovellusten käyttöönottoa konteilla. Sen avulla voidaan rakentaa, testata ja käyttöönottaa Linux ja Windows Serverin konttisovelluksia. Uudet ominaisuudet ja korjaukset saadaan tehtyä nopeasti ja ilman seisokkeja. Docker-kontti vastaa melkein hypervisoria ja niissä ajetaan erilaisia Linux ja Windows -sovelluksia. Sovellusten käynnistäminen on huomattavasti nopeampaa ja erillistä käyttöjärjestelmää ei tarvita. Nykyään Docker-konteja käytetään paljon sovelluskehityksessä ja niiden siirtäminen tuotantoon on todella nopeaa. Siksi haavoittuvuuksia täytyy pystyä skannaamaan samassa tahdissa kuin uuden koodin kirjoitus. Näin saadaan ajoissa tiedot mahdollisista tietoturva-aukoista ja voidaan korjata ne.

Opinnäytetyössä oli kaksi päätehtävää. Ensimmäisenä oli etsiä työkaluja tai ohjelmia, jotka voivat skannata tietoturvahaavoittuvuuksia valmiista Docker imagesta. Haavoittuvuusskannereita vertailtiin laadullisena tutkimuksena. Niiden täytyy kyetä skannaamaan omia Docker rekistereitä eikä pelkästään eri valmistajien omia kuten Docker Hubin Security Scanning. Haavoittuvuusskannerin tarkoitus on tarkastaa omien Docker-rekisterien imaget tietoturvahaavoittuvuuksista, jotka paikataan päivityksillä. Muiden rakennettujen imageiden skannausta voidaan myös suorittaa, ennen kuin niitä otetaan käyttöön. Haavoittuvuusskanneri myös integroidaan Jenkins CI/CD (Continuous Integration/Continuous Development) prosessiin, jossa valmis Docker image tarkastetaan ennen omaan rekisteriin työntämistä. Jos kriittisiä haavoittuvuuksia löytyy liikaa, Jenkins estää rekisteriin työnnön. Näin vältetään käyttämästä tietoturva-aukkoja sisältäviä imageita testaamisessa ja tuotannossa.

Toisena tehtävänä oli tutkia, miten saadaan konfiguroitua mahdollisimman tietoturvallinen Kubernetes-konttiympäristö. Konfiguroinnissa otettiin huomioon mm. Kubernetes, Docker ja verkotus.

1.2 Toimeksiantaja

Protacon Group perustettiin Jyväskylässä vuonna 1990, ja konserniin nykyään kuuluvat tytäryhtiöt Protacon Analyzes Oy, Protacon Solutions Oy sekä Protacon Technologies Oy (Historia n.d.). Protacon tarjoaa monipuolisesti palveluita eri aloilta mm. automaatiotekniikkaa ja digitalisaatiota. Protaconilla työskentelee yli 250 henkilöä, ja se on tehnyt asiakkuuksia jo yli 6000 yrityksen kanssa. (Protacon n.d.) Toimipisteitä on yksitoista, joista kaksi sijaitsee Jyväskylässä ja muut kymmenellä muulla Suomen paikkakunnalla sekä yksi Kiinassa (Yhteystiedot n.d.). Kuviossa 1 on Protacon Groupin nykyinen logo.

Kuvio 1. Protacon Groupin logo (Media n.d.)

Opinnäytetyön toimeksiantajana toimi Protacon Solutions Oy, joka tuli osaksi Protaconin konsernia vuonna 1998 (Historia n.d.). Protacon Solutions Oy tarjoaa ICT- ja ohjelmistopalveluita noin 100 työntekijän voimin (Protacon Solutions Oy n.d.). ICT-palveluihin kuuluu pilvipalveluiden ja työasemasovellusten tuki- ja ylläpitopalveluita, kapasiteettipalveluita, työasemien ja palvelimien asennuksia ja käyttöönottoa, tietoturvapalveluita, toimisto- ja ryhmätyösovelluksia esimerkiksi Microsoft Office 365 ja G Suite (ICT-palvelut n.d.). Yrityksen liikevaihto oli noin 8,0 miljoonaa ja tilikauden tulos noin 2,9 miljoonaa vuonna 2016. Yrityksellä on ollut kasvua vuodesta toiseen. (Protacon Solutions Oy n.d.)

2 Virtualisointi

2.1 Yleistä

Virtualisoinnilla pyritään luomaan useita virtuaalikoneita yhdellä fyysisellä tietokoneella tai serverillä, joita on eristetty toisistaan. Virtuaalikoneille määritetään omia resursseja fyysisestä koneesta kuten prosessoria ja muistia. Alkuperäistä fyysistä rautaa kutsutaan hostiksi eli isännäksi ja virtuaalikoneita guesteiksi eli vieraiksi. Yksinkertaisesti kerrottuna virtualisointi luo ympäristöjä ja resursseja, joita tarvitaan alikäytetystä raudasta. (Understanding virtualization 2017.)

Esimerkiksi kolme palvelinta tarjoaa omia palveluitaan: sähköpostia, verkkosivuja ja vanhoja sovelluksia. Ne käyttävät 30 % palvelimen resursseista, eli jokainen käyttää virtaa, jäähdytystä, ja niitä täytyy ylläpitää. Virtualisoinnilla sähköpostipalvelimelle voidaan siirtää vanhat sovellukset, jolloin yksi palvelin säästyy jäähdytys- ja ylläpitokustannuksista. Tämä oli yksi syy, miksi virtualisointi lähti kasvuun yrityksissä. (What is virtualization? 2017.)

Virtualisoitujen resurssien avulla järjestelmänvalvojat voivat jättää fyysisen asennuksen huomiotta. Päivittäminen tapahtuu saumattomasti virtuaalikoneiden tai sovellusten tietämättä pääkoneen muutoksista. Myös käyttökatkot pienenevät merkittävästi. (Understanding virtualization 2017.)

2.2 Hypervisor

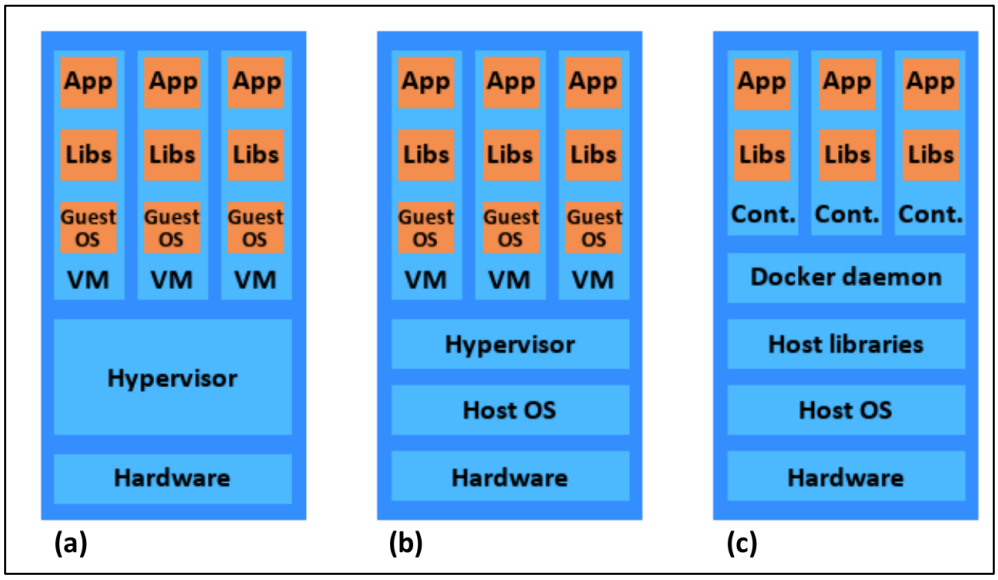
Hypervisor-tyyppejä on kaksi: Hypervisor type 1 ja 2. Hypervisor type 1 kutsutaan myös ‘bare metal’ hypervisoriksi, koska type 1 hypervisor asennetaan suoraan raudan päälle ’käyttöjärjestelmäksi’ ja se toimii yhdellä tasolla. Tämä on kevyempää kuin asentaisi hypervisorin toisen käyttöjärjestelmän päälle, koska tähän tarvitaan vähemmän resursseja palveluiden ajamiseen ja virtuaalikoneet saavat ne käyttöönsä. Type 1 hypervisorin alla voidaan ajaa virtuaalikoneita erilaisilla käyttöjärjestelmillä. Näihin koneisiin päästään käsiksi komentorivityökalulla tai erillisillä hallintasovelluksilla.

Type 1 hypervisoreita ovat esimerkiksi VMwaren ESXi Hypervisor, Microsoft Hyper-V ja XEN Hypervisor. (Daou 2016; Chesler 2017.)

Hypervisor type 2:ta kutsutaan ‘hosted’ hypervisoriksi, mistä type 2 saa nimensä kaksitasoisesta kokonaisuudestaan. Type 2 hypervisor on ohjelmisto, joka asennetaan fyysisen koneen käyttöjärjestelmässä ja hoitaa virtualisointinsa. Tätä voidaan hyödyntää monella tavalla kuten luomalla virtuaalikoneita, jossa testataan eri käyttöjärjestelmiä ja niissä erilaisia niille tarkoitettuja sovelluksia. (Daou 2016.) Tai ajamalla yrityskäyttöjärjestelmän päällä toista käyttöjärjestelmää ilman dual bootia. Type 2 hypervisoreita ovat esimerkiksi Oracle VirtualBox, QEMU ja VMware Workstation (Chesler 2017).

Yrityskäytössä suurin osa käyttää hypervisoreista tyyppiä 1, joista suosituimmat ovat VMware ESXi Hypervisor ja XEN Hypervisor. Kaikki tunnetut julkiset pilvet kuten Google, IBM/Softlayer ja Joyent eivät käytä hypervisoreita vaan kontteja. (Bernstein 2014, 81-82.)

Kuviossa 2 on kolme erilaista palvelinta. Palvelin (a) on tyypin 1 hypervisor, palvelin (b) on tyypin 2 hypervisor ja palvelin (c) on kontti. Palvelimelle (a) on asennettu hypervisor suoraan palvelinlaitteiston päälle, kun taas palvelimelle (b) on asennettu hypervisor isäntäkoneen käyttöjärjestelmän päälle. Molemmat palvelimet luovat virtuaalikoneet, joilla ovat omat käyttöjärjestelmät, kirjastot ja sovellukset. Palvelimella (c) on käyttöjärjestelmälle asennettu Docker daemon, joka ohjaa konttien luonnin. Konteilla ovat omat kirjastot ja sovellukset.



Kuvio 2. Eri virtualisointimalleja (Combe, Di Pietro & Martin 2016)

2.3 Docker

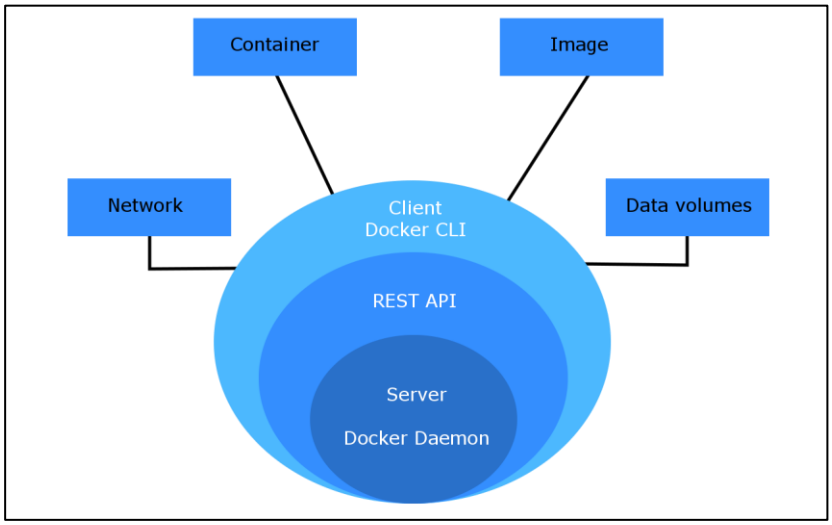
2.3.1 Yleistä

Docker on avoimen lähdekoodin projekti, joka tarjoaa järjestelmällisen tavan automatisoida nopeampaa Linuxin sovellusten käyttöönottoa konteilla (Bernstein 2014, 82). Dockerin avulla voidaan rakentaa, testata ja käyttöönottaa Linux ja Windows Serverin konttisovelluksia, jotka on kirjoitettu millä tahansa ohjelmointikielellä riskeittä yhteensopimattomuuksista tai versioiden ristiriidoista. Se vähentää 65 % aloitusajasta, kun ei tarvitse asennella ja ylläpitää erilaisia servereitä tai kehityskoneita, vaan päästään nopeasti ajamaan monimutkaisia monen kontin sovelluksia. Uudet ominaisuudet ja korjaukset saadaan tehtyä nopeasti ja ilman seisokkeja. Kaikki riippuvuudet ajetaan konteissa mikä vähentää toimimattomuusongelmia eri koneilla (ts. toimii joka koneella). (What is Docker 2017.)

Docker toimii Docker Enginellä, joka koostuu serverin Docker Daemonista, REST API:sta (Representational state transfer Application Programmable Interface) ja Docker CLI:stä (Command Line Interface). Docker Daemon on ohjelma tai prosessi palvelimella, joka luo ja hallitsee Docker objekteja kuten kontteja, imageita, verkkoa

ja dataa. REST API toimii Docker CLI:n ja Daemonin välillä kommunikoinnin kanavana. Se ottaa vastaan komentoja Docker CLI:ltä daemonille ja suorittaa ne. Docker CLI käyttää REST API:a vuorovaikuttaakseen Docker Daemonin kanssa CLI komennoilla tai skripteillä esimerkiksi *docker run*. (Docker overview 2017.)

Kuviossa 3 on havainnollistettu Docker Engine, jossa Docker Daemon on koko järjestelmän ydin, joka hallitsee objekteja. Seuraavana on REST API ottamassa vastaan pinnalla olevan Docker CLI:n komennot.



Kuvio 3. Docker Engine (Docker overview 2017)

2.3.2 Kontit

Docker-kontit ovat kevyitä ja eristettyjä virtuaalikoneen tapaisia sovellusten ajajia. Ne vievät tilaa vain muutaman kymmenisen megatavun, kun taas virtuaalikoneille asennetaan omat kokonaiset käyttöjärjestelmät. Docker-kontin sovellukset on eristetty toisistaan ja sen asennetussa käyttöjärjestelmässä, jolloin konttien viat eivät kulkeudu isäntäkoneelle. Docker kontteja voidaan ajaa kaikilla merkittävillä Linux ja-

keluilla, Microsoft Windowsilla ja millä tahansa infrastruktuurilla kuten virtuaalikoneilla, ”bare metalilla” ja pilvessä. Ne ovat todella nopeita käynnistymään, varsinkin kun Docker kontin image löytyy jo valmiiksi. (What is a Container 2017.)

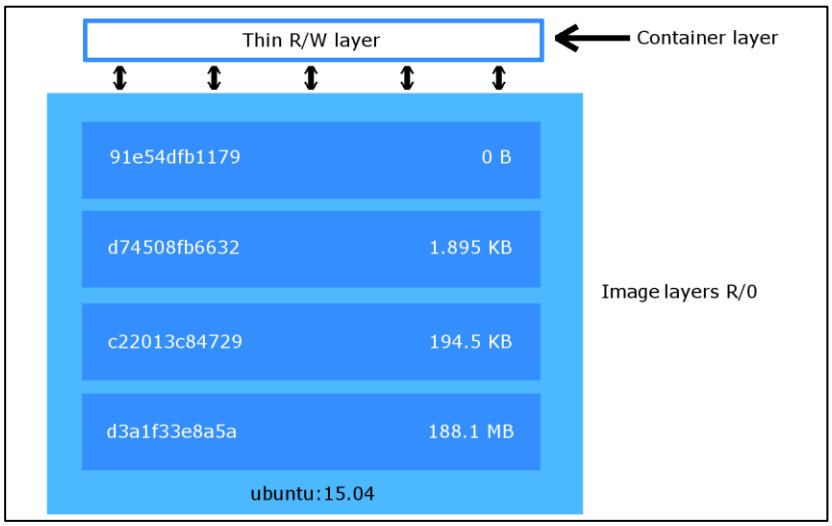
Konttien sovellukset jakavat käyttöjärjestelmän, ja lopputuloksena käyttöönotot ovat merkittävästi pienempiä kooltaan verrattuna hypervisorin käyttöönottoihin, mikä mahdollistaa satojen konttien ajamisen isäntäkoneella verrattuna virtuaalikoneiden rajoitettuun määrään. Koska kontit käyttävät isäntäkoneen käyttöjärjestelmää, kontin uudelleenkäynnistys ei käynnistä tai uudelleenkäynnistä isäntäkoneen käyttöjärjestelmää. (Bernstein 2014, 82.)

2.3.3 Imaget

Docker-kontit luodaan imageista, jotka voivat sisältää pelkän käyttöjärjestelmän perusteet tai koostua valmiiksi rakennetusta sovelluspinosta valmiina käynnistettäväksi. Kun rakennetaan imageita Dockerilla, jokainen käskytetty komento rakentaa uuden layerin eli kerroksen edellisen perään. Komennot suoritetaan manuaalisesti kontin sisällä olemassa olevasta valitusta imagesta ja tallentamalla uudeksi imageksi. (Bernstein 2014, 82-83; Combe, Di Pietro & Martin 2016, 56.)

Docker image ja layerit tunnistetaan yhteenvedolla algoritmi:heksa esimerkiksi *sha256:fc92eec5cac70b0c324cec2933cd7db1c0eae7c9e2649e42d02e77eb6da0d15f*. Heksa lasketaan sha256-algoritmilla ja sisällön muututtua lasketaan uudestaan. Tämän avulla tiedetään, onko ladattu oikea image rekisteristä. Rekistereistä kerrotaan luvussa x.x. Docker image koostuu järjestetyillä layerien yhteenvedoilla (Brown 2016.)

Kuviossa 4 on käytetty virallista ubuntu:15.04 Docker imagea pohjana jolla on layer ID d3a1f33e8a5a. Sen perään on tehty kolme eri käskytettyä komentoa, joten sillä on päällään kolme eri layeria lisää. Kun image otetaan käyttöön konttiin, container layeriin kirjoitetaan kaikki tulevat muutokset kuten uudet ja poistetut tiedostot sekä muutokset. Kun kontti poistetaan, container layer poistuu myös. (About images, containers, and storage drivers 2017.)



Kuvio 4. Docker imagen layerit eli kerrokset (About images, containers, and storage drivers 2017)

2.3.4 Dockerfile

Toinen vaihtoehto imagen rakennuksessa on sen automatisointi käyttämällä Dockerfilea. Kukin Dockerfile on skripti, joka koostuu erilaisista komennoista (ohjeista) ja argumenteista, jotka oikein listattuna automaattisesti tekevät toimintoja pohjaimageen luodakseen uuden imagen. Pohjaimagen lisäksi Dockerfile voi sisältää esimerkiksi avattuja portteja. Dockerfileja käytetään yksinkertaistamaan käyttöönottoprosessit alusta loppuun. (Bernstein 2014, 82-83; Combe, Di Pietro & Martin 2016, 56.)

Dockerfileen on myös mahdollista lisätä tiedostoja toiminnoilla ADD tai COPY haluamaansa polkuun isäntäkoneelta ja URL-osoitteista. Tämän avulla voidaan lisätä valmiita konfiguraatiotiedostoja. Toinen käytännöllinen toiminto on ENV eli ympäristömuuttuja, joita tarvitaan usein sovelluksissa. Docker tarkistaa ensin Dockerfilen että sen komennot on oikein asetettu ennen imagen rakennusta. Jos saman imagen rakennusta on tehty, voidaan käyttää välimuistista layeria aikaisemmasta käytetystä komennosta. Tämä nopeuttaa imageiden tekemistä, jos tarvitsee korjata jokin rakennuksessa käynyt vika. Dockerfilestä rakennetaan Docker image komennolla *docker build*. (Dockerfile reference 2017.)

Kuviossa 5 on esimerkki Dockerfilesta, jossa käytetään pohjana ubuntun Docker imagea, päivitetään järjestelmä apt-get updatella, asennetaan muutama ohjelma, luodaan kansio, avataan portti 5900 ja asetetaan CMD ajamaan kontin käynnistyksen parametrit.

```
FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc

# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd

# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900

CMD    ["x11vnc", "-forever", "-usepw", "-create"]
```

Kuvio 5. Dockerfile esimerkki (Dockerfile reference 2017)

2.3.5 Registry & repository

Docker imaget täytyy tallentaa registryihin eli rekistereihin esimerkiksi Docker Hubiin. Se on paikka, johon kehittäjät voivat työntää heidän Docker imageitaan ja käyttäjät voivat ladata niitä. Näitä kutsutaan registryiksi. Niitä voi olla julkisina näkyvillä ilmaiseksi tai maksua vastaa piilotettuna. Docker Hubista löytyy myös virallisia Docker imageita. (Combe, Di Pietro & Martin 2016, 56.)

Ne löytyvät julkisilta, sertifioiduilta repositoryilta eri valmistajilta ja avustajilta. Docker imageita ovat tehneet ainakin Canonical, Oracle ja Red Hat, joiden imageita voidaan käyttää sovellusten ja palveluiden rakentamiseen. (Overview of Docker Hub 2017.) Imagen päivitykset, konfiguraatiomuutokset ja rakennushistoriat pysyvät tallessa registryissä (What is Docker 2017).

Repository toimii Docker imagen tekijän ja säilytyspaikan nimenä. Ensimmäinen osa kertoo käyttäjän nimen ja toinen puolestaan imagen sisällön. Esimerkiksi ansible/centos7-ansible on ansible-nimisen käyttäjän Docker image, jossa ansible-ohjelma on asennettu Centos 7 -käyttöjärjestelmälle. Viralliset Docker repot eivät sisällä käyttäjän nimeä ollenkaan, mistä tietää niiden virallisuuden. (Repositories on Docker Hub 2017.)

Tagien avulla voidaan nimetä Docker imagen repository ja sen perässä kaksoispisteen jälkeen versio. Versiointitapa on täysin vapaa esimerkiksi versioimalla päivämäärän perusteella ja/tai imagen kuuluvan testaukseen *dev:v1.6.14*. Tagaaminen onnistuu komennolla *docker tag IMAGE ID image/tag,* jossa image ID on alkuperäinen Docker imagen nimi tai ID ja image/tag on uusi imagen nimi. Esimerkiksi *docker tag ubuntu:latest ubuntu:v1.6.14.* (Wallen 2017.)

Docker push työntää Docker imagen registryyn ja *docker pull* puolestaan hakeen imagen registrystä (Wallen 2017).

3 Kubernetes

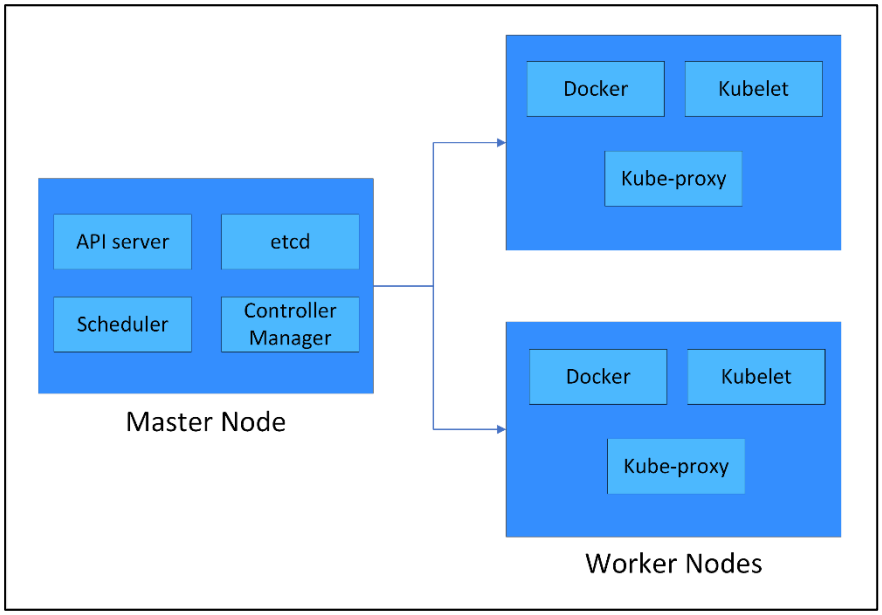
3.1 Yleistä

Google ilmoitti Kuberneteksen kesäkuussa 2014, ja sitä alkoivat kannattaa suuri määrä yrityksiä. Start-up-yrityksistä CoreOS, MesoSphere ja SaltStack sekä kuuluja yrityksiä Microsoft, VMware, IBM, Red Hat, Googlen Cloud ja Container Enginet. (Bernstein 2014, 84.)

Kubernetes on avoimen lähdekoodin järjestelmä, joka hoitaa konttisovellusten automaattista käyttöönottoa eli deploamista, skaalausta ja hallintaa. Sillä on nopeaa ja luotettavaa käyttöönottaa sovelluksia, skaalata niitä lennosta, ottaa käyttöön uusia ominaisuuksia saumattomasti ja rajoittaa suorituskyvyn käyttöä valituilta resursseilta. Se hyödyntää konttiteknologiaa fyysisten tai virtuaalisten koneiden klustereissa. (What is Kubernetes? n.d.)

Kubernetes klusteri koostuu yksittäisestä master nodesta, jonka tarkoitus on hallita Kuberneteksen kokonaisuutta, ja vähintään yhdestä worker nodesta, jotka ajavat haluttuja sovelluksia. Master node koostuu kolmesta prosessista: kube-apiserver, kube-controller-manager ja kube-scheduler. Kube-apiserverillä kommunikoidaan ja suoritetaan operaatioita Kubernetes klusterilla, kube-controller-manager hoitaa klusteritason tehtävät kuten prosessien replikoinnin tai ylläpitää worker nodeja ja kube-scheduler on vastuussa sovellusten ajoituksiin worker nodeille. Master node sisältää myös etcd:n, jossa säilötään koko klusterin konfiguraatio. (Concepts n.d.;Lukša 2017, 16)

Worker nodeilla ovat deplotut sovelluskontit. Kun käyttäjä deploaa listan sovelluksia, Kubernetes deploaa ne jollekin klusterin worker nodelle. Worker node sisältää konttiohjelmiston, kubeletin ja kube-proxyn prosessit. Konttiohjelmistona toimii Docker, rkt tai jokin muu konttien ajamiseen soveltuva ohjelma. Kubelet kommunikoi master nodelle ja ohjaa kontteja. Kube-proxy toimii sovelluskomponenttien verkkoliikenteen välityspalvelimena ja kuormantasaajana. (Lukša 2017, 15-17) Kuviossa 6 havainnollistettu master ja worker nodet sekä niiden sisältämät komponentit.



Kuvio 6. Master ja worker nodet prosesseineen (Lukša 2017, 18)

Kubernetes klusterin voi asentaa moneen paikkaan: omalle tietokoneelle, yrityksen palvelimelle tai pilvipalveluiden tarjoamille palvelimille esimerkiksi Google Compute Enginelle. Omalla tietokoneella on helppoa ja huoletonta testata yhden noden Minikubella, joka sisältää paljon Kuberneteksen ominaisuuksia. (Lukša 2017, 32)

Kubernetesta hallitaan kubectl komentorivityökalulla, jolla on mahdollista luoda, muokata ja poistaa objekteja sekä saada niistä tietoa. Kubectl:n jälkeen laitetaan haluttu komento eli operaatio, joka suoritetaan objektille. Tämän jälkeen tulee objektin nimi ja valinnaiset liput. (Overview of kubectl n.d.) Yksinkertaisesti esitettynä komento näyttää tältä:

kubectl (komento) (objekti) (nimi) (liput)

Esimerkiksi komennolla *"kubectl get pods -n testnamespace"* saadaan lista podeista namespacesta testnamespace. Mainitsemalla nimen saa tietoa juuri siitä objektista (Overview of kubectl n.d.). Objektien luonnissa täytyy kertoa teknisiä tietoja, mikä voi johtaa pitkään komentorivikomentoon. Näiden luomista voi helpottaa kirjoittamalla YAML (YAML Ain't Markup Language)-tiedoston, joka sisältää luontiin tarvittavat tiedot. (Understanding Kubernetes Objects n.d.) Kuviossa 7 on esimerkkinä YAML-tiedosto, jossa luodaan podi nimellä nginx-pod namespaceen testispace3 ja sen sisällä on nginx kontti.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: testispace3
spec:
  containers:
  - name: nginx-container
    image: nginx:latest
```

Kuvio 7. Kubernetes objektin YAML-tiedosto

3.2 Objektit

3.2.1 Yleistä

Kubernetes edustaa tilaansa objekteilla, jotka voivat kertoa: mitä konttisovelluksia on ajossa, mitä resursseja sovelluksilla on käytössä ja millä politiikalla sovellukset käyttäytyvät kuten päivitykset, vikatilanteet ja uudelleenkäynnistykset. Objekteja luodaan, muokataan ja poistetaan Kubernetes API:n kautta. (Understanding Kubernetes Objects n.d.)

Objektien organisointia voi tehdä labelien eli merkkien avulla. Ne ovat key-value tyylisiä pareja, joita voi liittää mihin tahansa resurssiin ja näin ryhmitellä niitä. Objekteilla voi olla enemmänkin kuin yksi label, kun key on uniikki. Labeleita voi lisätä objektien luontivaiheessa tai myöhemmin tarpeen tullen ilman objektin uudelleenkäynnistämistä. (Lukša 2017, 59)

3.2.2 Pod

Podi on yksi Kuberneteksen käytetyin objekti, joka sisältää yhden tai useamman kontin samalla worker nodella. Jokainen podi on kuin looginen kone omalla IP-osoitteella, hostnimellä, prosesseilla, jne. (Lukša 2017, 37) Podin sisällä olevat kontit voivat kommunikoida toistensa kanssa käyttäen IP-osoitteena localhostia. Ne voivat myös käyttää samaa podin asettamaa tiedostojärjestelmää. (Pod Overview n.d.)

Pelkän podin luominen ei ole suositeltavaa, koska se poistetaan virheiden tapahtuessa esimerkiksi noden kaatuessa tai prosessin epäonnistuessa. Podit eivät pidä itsestään huolta tai osaa itsestään kuormantasausta useammalla podilla, minkä takia käytetään toista Kuberneteksen objektia pitämään ne halutulla määrällä päällä. Tämä objekti on kontrolleri, joka sisältää podin tekniset tiedot, niiden lukumäärän ja virheiden tapahtuessa korjaa ne itsestään. Kontrollereita on erilaisiin käyttötarkoituksiin esimerkiksi Deployment, StatefulSet ja DaemonSet. (Pod Overview n.d.)

3.2.3 Service

Jokaisella podilla on omat IP-osoitteet ja terminoituessa ne saavat uudet. Servicen avulla podeihin saadaan yhteys muutoksista huolimatta. (Services n.d.) Service saa luonnin yhteydessä IP-osoitteen, joka ei vaihdu olemassaolonsa aikana. Käyttäjät eivät yhdistä suoraan podeihin vaan liikennöidään serviceiden kautta niihin. (Lukša 2017, 42) Yhdistäminen yleensä tehdään labelien avulla, jotka laitetaan objektien teknisiin tietoihin esimerkiksi app=mysql. Label on podissa ja service yhdistää sinne samalla labelillaan. Service sisältää myös avattavan portin, jolla podiin pääsee yhdistämään. (Services n.d.)

Service tyyppejä on monenlaisia ja yleisimmät ovat: ClusterIP, NodePort ja LoadBalancer. ClusterIP on Servicen oletusarvo ja avaa pääsyn vain klusterin sisällä oleville objekteille. NodePort avaa pääsyn jokaisella noden IP-osoitteella tietyllä kiinteällä portilla. LoadBalancer avaa pääsyn ulkopuolisille käyttäen pilvipalvelun tarjoajan kuormantasausta. (Services n.d.)

Jokainen service saa myös DNS-tiedon Kuberneteksessä pyörivän kube-dns podin avulla. Kube-dns hoitaa Kuberneteksen DNS-kyselyt, kun se tietää olemassa olevat servicet. Esimerkiksi jos servicen nimi on palvelu ja se sijaitsee namespacessa nimiavaruus, saadaan siihen yhteys Kuberneteksessä *palvelu.nimiavaruus.svc.cluster.local*, mutta pelkkä *palvelu.nimiavaruus* riittää samassa klusterissa. (Lukša 2017, 103-104)

3.2.4 Namespace

Namespacet tulevat hyödyllisiksi, kun halutaan erotella objekteja erillisiin ei-päällisiin ryhmiin. Tämä auttaa organisoimaan resursseja moniin erilaisiin ympäristöihin esimerkiksi tuotanto ja kehitys. Silloin voi myös objektien nimen olla samanlaiset, koska DNS-tiedot ovat erilaiset namespacen avulla. Jos namespacea ei käytetä ollenkaan, objektit luodaan automaattisesti default nimiseen namespaceen. Kuberneteksen luomat objektit, kuten kube-dns, sijaitsevat namespacessa kube-system. (Lukša 2017, 64)

Namespace ei kuitenkaan eristä objektejaan muista namespacen objekteista. Jos namespace X tietää namespace Y:n podin IP-osoitteen, mikään ei estä lähettämästä liikennettä kuten HTTP Requestia. Namespacen eristämiseen tarvitaan muita työkaluja. (Lukša 2017, 66)

3.2.5 ServiceAccount

Kubernetes erottelee API:llensa yhdistävät käyttäjät kahdeen ryhmään: ihmiset eli oikeat käyttäjät ja podit (tarkemmin sanottuna niiden sovellukset). Service Accountit ovat tarkoitettu podien käytettäviksi ja yksi löytyy jokaisesta namespacesta. Namespacen luonnin yhteydessä luodaan sinne samalla Service Account ja niitä voi luoda itse lisää. Podin määrityksiin on lisätty oletus Service Account automaattisesti, jos sille ei ole laitettu itse mitään. Service Account saa oikeutensa podiin liitetystä Secret tokenista, joka löytyy polusta */var/run/secrets/kubernetes.io/serviceaccount*. (Lukša 2017, 267-268)

3.2.6 Secret

Docker-imageen on mahdollista laittaa konfiguraatiota ympäristömuuttujien kautta nopeasti ja helposti verrattuna tiedostojen lisäämistä erikseen. Kuberneteksessä voi myös lisätä ympäristömuuttujia suoraan komentoriviltä tai YAML-tiedostoon. Mutta jos konfiguraatio on sisältää arkaa tietoa, sitä ei ole hyvä säilyttää siellä. (Lukša 2017, 153)

Secretin tarkoitus on säilyttää arkaa tietoa kuten salasanoja, tokeneita ja ssh-avaimia. Sitä voidaan käyttää ympäristömuuttujana, tekstinä ja tiedostona. Sen voi luoda suoraan komentoon kirjoitetusta tekstistä tai tiedostoista. Ensin Secretin data täytyy enkoodata base64:lla ennen kuin se voidaan laittaa YAML-tiedostoon ja luoda siitä Secret. Luonnin jälkeen Secrettiä voidaan käyttää esimerkiksi podin määrityksessä referoimalla. Secretin dataa ei saa suoraan näkyville kubectl get tai kubectl describe komennoilla tarkoituksena suojata niiden näkyminen. ConfigMapilla saa myös konfiguraatiodataa, mutta sen data ei ole salattu. (Secret n.d.)

3.3 Google Container Engine

Google Cloud Platform tarjoaa GKE (Google Container Engine) palvelua, joka on heidän ylläpitämä Kubernetes. Google Site Reliability Engineerit ylläpitää Container Engineä täysin itse, että omat klusterit pysyvät päällä ja ajan tasalla, ja samalla heidän iso turvatiimi on turvaamassa kovilla standardeillaan. Kubernetekseen tuodaan jatkuvasti uusia versioita, joita voi automaattisesti päivittää klustereihin. GKE pyörii Googlen omalla kovennetulla käyttöjärjestelmällä, joka on optimoitu konteille. Klustereiden käyttäjien hallinta onnistuu Google tunnuksilla ja roolituksilla. (Container Engine n.d.)

4 Tietoturva

4.1 Yleistä

Tietoturvasta kirjoitetaan uutisotsikoita yhä useammin, oli kyse sitten turvattomista IoT (Internet of Things) -laitteista tai haavoittuvuuksien kautta sairaaloiden järjestelmiin tarttuvista madoista (Takala 2016; Hall 2017). Iso-Britannian sairaaloiden järjestelmiä meni nurin, koska siellä oli vanhoja päivittämättömiä koneita. WannaCrypt/WannaCry vaatii maksamaan kryptatuista tiedoista Bitcoineja, muuten tiedostot jäävät ikuisesti lukkoon (Hall 2017). Madon levittäjää on vaikea saada selville, kun virtuaalivaluutan käytössä ei tiedetä vastaanottajaa. Nämä ovat vain muutamia esimerkkejä, miksi kannattaa päivittää tietoturvahaavoittuvuuksien varalta ja pysyä aktiivisena tietoturvassa.

Tietoturvaa tulee jokaiselle vastaan päivittäin, kun eri palveluihin kysytään tunnuksia huijausviesteillä sähköpostin kautta (Suomalaisten tietoja kalastellaan aktiivisesti 2017). Käyttäjän täytyy itse käyttää järkeä ja miettiä hetki, onko saatu viesti aito tai onko jokin tarjous liian hyvä ollakseen totta? Käyttäjä on tietoturvan heikoin lenkki, ja henkilöstöä on siksi myös hyvä kouluttaa näitä varten (Mehiläinen 2017).

Tietoturvasta on hyvä lähteä liikkeelle CIA-mallista, josta kerrotaan seuraavassa luvussa.

4.2 CIA-malli

CIA-malli (Confidentiality, Integrity, Availability) on vanha ja klassinen malli kertoa tietoturvan pääkohdat. Niitä on kolme, ja jokaisella ovat omat tehtävänsä tarjota tietoturvaa (Confidentiality, Integrity, and Availability 2016).

Confidentiality eli luottamuksellisuus perustuu tiedon säilyttämiseen luottamuksellisena, että siihen ei pääse käsiksi luvattomat tahot vaarantaen muiden tietoon pääsyn. Esimerkiksi pankkitilin omistaja ja pankin työtekijät pääsevät pankkitilille mutta muut eivät vahingossa tai tahallaan. (Confidentiality, Integrity, and Availability 2016.) Yhtä hyvin joku voi lukea muiden sähköpostiviestejä, kun on pääsy sähköpostin käsittelyä tekevään laitteistoon ja verkkoon. Siksi luottamuksellisuus edellyttää myös tiedon ja sen kuljettamisen suojausta. Confidentialityn synonyymeja ovat privacy ja secrecy, jotka viittaavat yksityisyyteen ja salassapitoon. (Kerttula 2000, 93-95)

Integrity eli tiedon eheys tarkoittaa tiedon säilyttämistä alkuperäisenä ja muuttumattomana tietoliikenteessä (Confidentiality, Integrity, and Availability 2016). Tämä edellyttää myös tiedon pysymistä alkuperäisenä laitteiston tai ohjelmiston vikaantuessa. Muutoksia suorittavat vain siihen oikeutetut tahot, ja muutostiedot pidetään tallessa historiatietona. (Kerttula 2000, 93) Esimerkiksi webselaimella mentäessä johonkin Internet-osoitteeseen hyökkääjä ohjaakin väärennetylle samannäköiselle websivulle, jolloin haettu websivu ei ollut alkuperäinen ja muutettu sivullisen tahosta (Confidentiality, Integrity, and Availability 2016).

Availability eli saatavuus tai käytettävyys tarkoittaa tiedon tai palvelun ollessa saatavilla niille oikeutetuille käyttäjille. Esimerkiksi pilvipalvelussa on käytettävissä omat tiedostot tarpeen vaatiessa ja vain sille kuuluvalle henkilölle. (Confidentiality, Integrity, and Availability 2016.) Hyökkäyksiä voidaan estää ainakin autentikoinnin tai tiedon salauksen käytöllä, että tietoon tai palveluun pääsisi pelkästään sen oikeat käyttäjät. Myös fyysinen suojaus pitää ottaa huomioon että fyysisille tiloille tai laitteille eivät pääse sivulliset. (Kerttula 2000, 97)

4.3 CVE

CVE (Common Vulnerabilities and Exposures) on julkisesti tunnettujen kyberturvallisuuden haavoittuvuuksien yhteisten nimien sanakirja. Nämä CVE-nimet helpottavat tiedon jakamista eri tietoturvakannoista ja -työkaluista sekä tarjoavat perustan organisaation tietoturvatyökalujen kattavuuden arvioimiseksi. Jos jokin raportti sisältää CVE-tunnisteet, voidaan nopeasti ja tarkasti katsoa korjausmetodit yhdestä tai useammasta erillisestä CVE-yhteensopivasta tietokannasta ongelman korjaukseen. CVE:t pisteytetään haavoittuvuuden vakavuuden mukaan niin huomataan, onko haavoittuvuus lievä vai vakava. CVE on julkisesti ladattavissa ja käytettävissä ilmaiseksi. (About CVE 2017.)

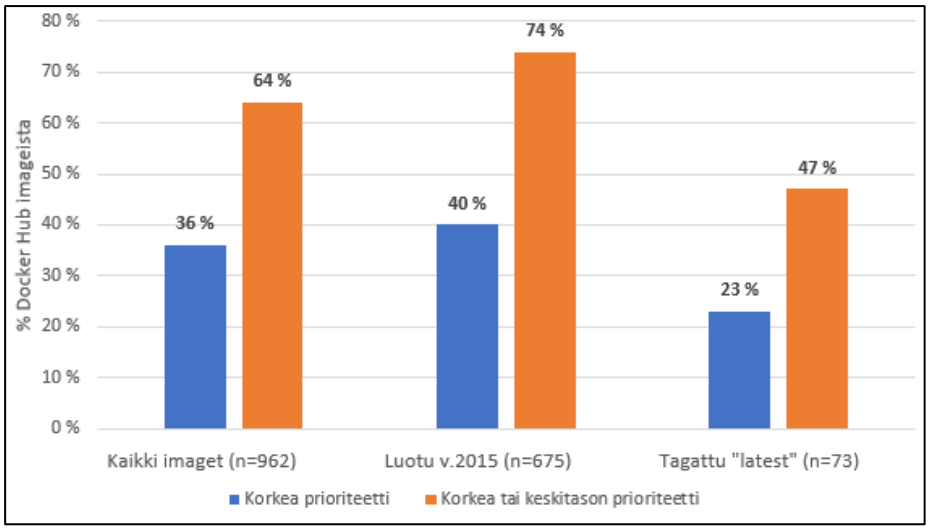
Ennen CVE:n julkaisua yrityksillä ja laitevalmistajilla olivat omat tietokannat joiden data oli vaihtelevaa ja erilaista. Haavoittuvuuksilla oli omat nimensä ja kuvauksensa, mikä sekoitti korjausten etsimistä. CVE:n tarkoitus on ratkoa nämä ongelmat standardilla. (About CVE 2017.)

CVE-tunniste koostuu CVE-etuliitteestä, vuosiluvusta ja juoksevasta numerosta. Tämä juokseva numero on neljästä seitsemään, joten ensimmäisten haavoittuvuuksien numerointi alkaa nollilla (CVE ID Syntax Change 2016). Esimerkiksi CVE-2014-0015 on vuodelta 2014 juoksevalla numerolla 15. Tällä CVE-tunnisteella kerrotaan libcurlin väärin uudelleenkäyttävän yhteyksiä, kun käytetään NTLM-autentikointia, mikä voi johtaa tahattomien kredentiaalien käyttämiseen ja altistaa herkkää tietoa. Korjausohjeina oli päivittää libcurl uudempaan versioon. (USN-2097-1: curl vulnerability 2014.)

4.4 Docker tietoturva

Toukokuussa 2015 BanyanOPSin tekemässä tutkimuksessa tuli ilmi, että yli 30 % Docker Hubin imageista virallisissa repositoryissa ovat hyvin alttiita erilaisille iskuille kuten Shellshockille ja Heartbleedille. Kun lisätään muiden käyttäjien imaget mukaan, luku kasvaa noin 10 %. Docker Hubissa oli 75 virallista repositorya tutkimuksen teon aikana. Kuviossa 8 on diagrammi virallisten imageiden haavoittuvuuksien määristä,

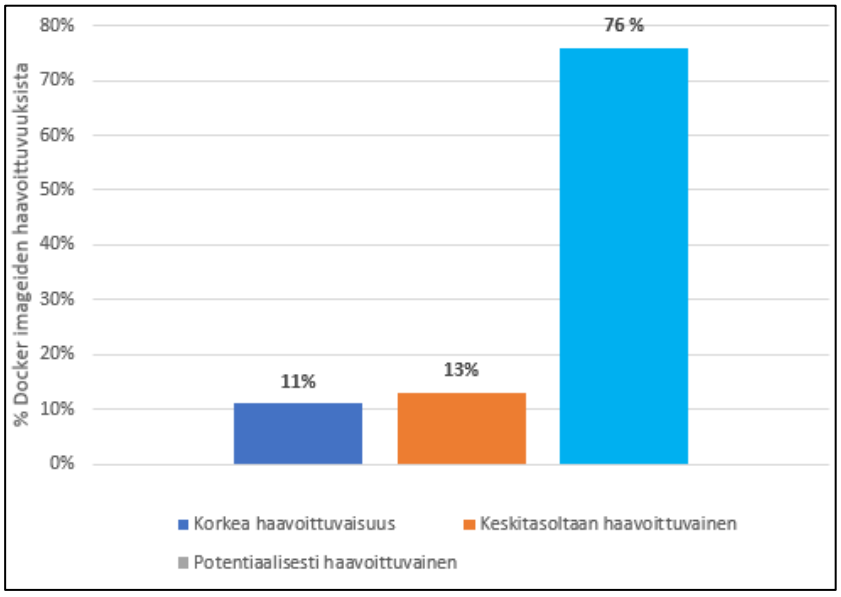
missä huomataan olevan korkea prioriteetin haavoittuvuuksia yli 36 % kaikista imageista. Latest tagilla olevilla imageita on 23 %, mitä yleensä käytetään omien imageiden pohjana. Jos katsoo luotujen imageiden määrää vuodelta 2015, korkean prioriteetin haavoittuvuuksia löytyy 40 % imageista. (Desikan, Gummaraju & Turner 2015.)



Kuvio 8. Virallisten imageiden haavoittuvuus määriä (Desikan, Gummaraju & Turner 2015)

Imaget pitäisi skannata ei pelkästään käyttöjärjestelmätasolla mutta myös applikaatiotasolla. Skannaus pitäisi olla integroituna jatkuvan kehityksen järjestelmässä, että voidaan pysyä hyvässä turvallisuustasossa (Desikan ym. 2015).

Maaliskuussa 2017 Federacy skannasi 91 virallista Docker repositorya. Skannauksessa tuli ilmi, että 24 % testatuista Docker imageista sisältää merkittäviä haavoittuvuuksia eli vakavuudeltaan korkean ja keskitason luokkaa. Kuviossa 9 on virallisten imageiden vakavuusjakauma, jossa 11 % Docker imageiden haavoittuvuuksista on vakavuudeltaan korkeita, 13 % vakavuudeltaan keskitasoa ja 76 % potentiaalisesti haavoittuvaisia (Sulinski 2017).



Kuvio 9. Virallisten imageiden vakavuusjakauma (Sulinski 2017)

Haavoittuvaisten Docker imageiden käyttäminen tuotannossa voi vaarantaa sekä yrityksen että asiakkaiden nimeä, ja siksi julkisesti näkyvillä olevien haavoittuvuuksien korjaaminen olisi ensimmäinen askel. Haasteena on kuitenkin, että miten käytännössä suoritettaisiin jatkuva päivittäminen. Haavoittuvuuksien poistaminen voidaan suorittaa pakettien päivittämisellä imagen rakentamisen aikana tai sen ollessa käynnissä. Imagen skannaus kannattaa suorittaa imagen rakennuksen jälkeen. Onneksi Docker imageiden skannaaminen on merkittävästi helpompaa kuin ennen. Esimerkiksi Docker Hub ja Quay.io tarjoavat imageiden skannausta niiden repositoryissaan avoimen lähdekoodin skannereilla. (Sulinski 2017.)

4.5 OWASP Top 10

OWASP (The Open Web Application Security Project) on maailmanlaajuinen voittoa tavoittelematon hyväntekeväisyysjärjestö, joka on keskittynyt ohjelmistojen turvallisuuteen. Sen tavoitteena on tehdä ohjelmistojen turvallisuudesta näkyvää, että yksilöt ja organisaatiot voivat tehdä perusteltuja päätöksiä OWASPin puolueettomalla

tiedolla. Materiaalit ovat saatavilla ilmaiseksi avoimella ohjelmistolisenssillä. (Welcome to OWASP 2017.)

OWASP Top 10 edustaa laajaa yhteisymmärrystä websovellusten kriittisistä tietoturvariskeistä. Projektin jäseniin kuuluu joukko tietoturva-asiantuntijoita eri puolilta maailmaa, jotka ovat jakaneet heidän asiantuntemuksiaan luettelon luonnissa. Listoja löytyy vuosilta 2004, 2007, 2010 ja 2013, mutta vuoden 2017 on vielä tekemättä odottaen rakentavaa kommenttia ja niiden analysointia. (Category:OWASP Top Ten Project 2017.)

Vuoden 2017 listan ehdokkaina ovat aikaisversioissa seuraavat:

1. Injection
2. Broken Authentication
3. Cross-Site Scripting (XSS)
4. Broken Access Control
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Insufficient Attack Protection
8. Cross-Site Request Forgery (CSRF)
9. Using Components with Known Vulnerabilities
10. Underprotected APIs (OWASP Top 10 Application Security Risks – 2017 2017).

Listan nro 9: *Using Components with Known Vulnerabilites* eli haavoittuvuuksia sisältävän komponentin käyttäminen on työhön liittyvin otsikko. Hyökkääjät tunnistavat heikon komponentin skannauksen tai manuaalisen analyysin kautta. Ne muokkaavat komponentin hyväksikäytettävyyttä niin, että voivat tehdä hyökkäyksen. Jos käytettävä komponentti on syvällä sovelluksen koodissa, hyväksikäytettävyys on vaikeampaa. Monet applikaatiot ja API:t (Application Programmable Interface) käyttävät kaikenlaisia riippuvaisuuksia ja kirjastoja, joista jokaisella applikaation tekijällä ei ole tietoa. Nämä voivat sisältää hyväksikäytettäviä aukkoja hyökätä. (Top 10 2017-A9-Using Components with Known Vulnerabilities 2017.)

Haavoittuvuutta ei välttämättä lähetetä CVE-tietokantoihin riippuen komponentista. Haavoittuvuuksista selville saaminen voi vaatia etsimistä tietokannoista ja sähköpostilistoille ilmoittautumista. Haavoittuvuuksien löytämiseen on onneksi automaattisia tapoja kuten komponentin jatkuva monitorointi, joka pitää tietoturvaa yllä. Lopuksi pitää tarkistaa, täytyykö komponentti päivittää. (Top 10 2017-A9-Using Components with Known Vulnerabilities 2017.)

5 Continuous Integration, Delivery, Deployment

5.1 Yleistä

Ohjelman kehittäminen ja julkaiseminen on monimutkainen prosessi, koska kehitys, testaus ja julkaiseminen täytyy olla nopeaa ja johdonmukaista. Kehittäjät ja organisaatiot ovat luoneet kolme erillistä strategiaa hallitakseen ja automatisoidakseen nämä prosessit (Ellingwood 2017).

Continuous Integration eli jatkuvan integroinnin tarkoitus on saada jatkuvasti koodia monelta kehittäjältä yhteen repositorion haaraan eli branchiin. Kun koodia saadaan ajoissa yhteen branchiin, voidaan eliminoida ennakkoon integroinnissa olevat bugit ja viat. Jotta tämä strategia toimii tehokkaasti, koodi täytyy testata useasti ja korjata ilmenneet ongelmat heti. Jos koodia syntyisi monelta kehittäjältä eristetysti silloin tällöin, ongelmia ja yhteensopivuuksia voi ilmentyä enemmän lopussa. (Ellingwood 2017.)

Continuous Delivery eli jatkuva toimitus laajentaa CI:tä automatisoimalla ohjelman toimittamista tuotantovalmiiksi. Kehitystiimi voi tarkistaa ohjelman koodin ja toimittaa nykyisen version tuotantoon pipelinen eli putkiston kautta. Se koostuu stageista eli vaiheista, jotka sisältävät eri testejä. Jos jokin vaihe epäonnistuu, kehitystiimi huomaa sen ja korjaa ilmenneet viat. Vaiheiden suoriututtua ohjelma on tuotantoon toimittamista lähempänä. (Ellingwood 2017.)

Continuous Deployment eli jatkuva käyttöönotto laajentaa CD:tä vielä enemmän ohjelman käyttöönotolla, kun se on läpäissyt eri testit putkiston vaiheiden läpi. Tämä mahdollistaa automaattisen ohjelman käyttöönoton, että kenenkään ei tarvitse manuaalisesti laittaa niitä itse. Myös ohjelmiston päivitykset voidaan asettaa, nopeasti että saadaan pienet ominaisuudet nopeasti asiakkaille. Kehitystiimi voi pysyä myös paremmin ajantasalla, miten ohjelma suoriutuu ja mitä sen versiota käytetään paraikaa. Kehitystiimin koodi täytyy olla ajantasainen ja suunniteltu tätä varten, että se ei riko mitään. (Ellingwood 2017.)

5.2 Jenkins

Jenkins on itsenäinen avoimen lähdekoodin automaatiopalvelin, jota voidaan käyttää automatisoimaan kaikenlaisia tehtäviä kuten ohjelman rakentamista, testausta ja käyttöönottoa. Se voidaan asentaa alkuperäisillä järjestelmäpaketeilla, Dockerilla tai suorittaa itsenäisesti millä tahansa koneella, jossa on asennettuna Java (Jenkins Documentation n.d.). Jenkinsiä voidaan käyttää CI/CD-palvelimena ja laajentaa sitä asentamalla sadoista eri lisäosista tarpeellisia työkaluja, jotka ovat helppo konfiguroida webhallinnan kautta. Jenkinsillä voidaan nopeasti jakaa tehtäviä eri koneille ja alustoille (Jenkins n.d.).

Aikaisemmin käyttäjät joutuivat tekemään manuaalisesti Jenkins työt eli jobit ja täyttämään ne asetuksilla ja tiedoilla webhallintasivun kautta. Kaiken tämän tekeminen manuaalisesti vaatii vaivannäköä käyttäjiltä, kun joka projektille täytyy tehdä koodin testit ja rakennukset. Pipeline pluginin eli lisäosan avulla käyttäjät voivat implementoida projektin build/test/deploy putkiston Jenkinsfileen. (Pipeline as Code with Jenkins n.d.)

Jenkinsfile on tiedosto, johon kirjoitetaan putkiston koodi. Sitä pidetään versionhallinnassa muutoksien kirjoitusta varten. Kuviossa 10 on esimerkki Jenkinsfilen sisällöstä, jossa node on kone tai alusta. Se luo myös workspacen eli työskentelytilan, jossa haettuja versionhallinnan tiedostoja käytetään. Staget eli vaiheet havainnollistavat eri putkiston vaiheita. Niiden sisällä voi olla erilaisia shell komentoja ja muiden pluginien toimintoja kuten junit. (Pipeline n.d.)

```
node {
    stage('Build') {
        sh 'make'
    }

    stage('Test') {
        sh 'make check'
        junit 'reports/**/*.xml'
    }

    stage('Deploy') {
        sh 'make publish'
    }
}
```

Kuvio 10. Jenkinsfile esimerkki (Pipeline n.d.)

6 Haavoittuvuusskannerit

6.1 CoreOS Clair

6.1.1 Yleistä

Clair on CoreOS:n avoimen lähdekoodin projekti, joka tekee staattista analyysiä haavoittuvuuksien etsimiseen applikaatiokonteista. Sen tehtävä oli tuoda läpinäkyvämpää näkökulmaa konttiympäristöjen tietoturvasta, joten Clair sai nimensä ranskalaisesta termistä tarkoittaen selkeää, kirkasta ja läpinäkyvää. (Clair 2017.)

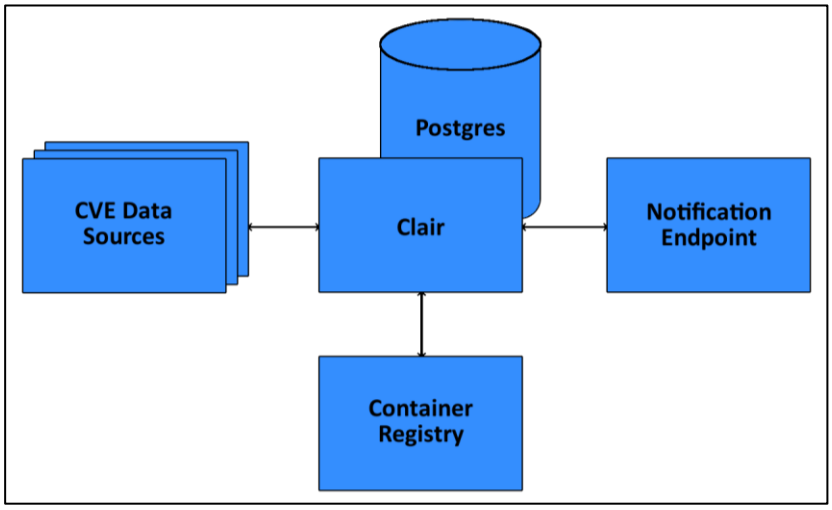
Se koostuu kahdesta palvelusta: PostgreSQL-tietokannasta ja Go-ohjelmointikielellä kirjoitetusta Clair API:sta. Tietokantaan ladataan CVE-tietoja kuudesta eri lähteestä:

- Debian Security Bug Tracker
- Ubuntu CVE Tracker
- Red Hat Security Data
- Oracle Linux Security Data
- Alpine SecDB
- NIST NVD. (Clair 2017.)

Ensimmäinen CVE-tietojen lataus kestää useita minuutteja ja seuraavat kerrat lyhyemmän ajan, koska tietokanta täytetään vain muokatuilla ja uusilla CVE-tiedoilla. Käyttäjät lähettävät Docker imagen kerrokset Clair API:lle, joka tekee listan imagelle

asennetuista käyttöjärjestelmän ohjelmistopaketeista ja säilyttää ne tietokannassa. Kerrosten lähettämisen jälkeen voidaan kysyä Clair API:lta tulokset heti koska korreloitu data käydään reaaliajassa eikä välimuistissa. Muuten Docker imagen kerrokset pitäisi lähettää uudestaan. (Clair 2017.)

Kuviossa 11 on Clairin perusympäristö, jossa Clair ja Postgres-tietokanta toimivat yhdessä, että se perusteisesti toimii. Clair hakee CVE Data Sources eli CVE-tietoja ja vie ne Postgres-tietokantaan talteen. Tarkastuksissa käytetään Container Registryä, jossa sijaitsevat Docker imaget. Notification Endpoint on rajapinta, johon voidaan lähettää viesti webhookilla, kun jokin CVE-tieto on päivittynyt, esimerkiksi CVE-tietoon tuli lisätietoa korjauskehotuksesta tai haavoittuvuuden vakavuus muuttui.



Kuvio 11. Clairin perusympäristö (Clair 2017)

Clairista löytyy yrityksille suunnattu versio, mutta se on integroitu CoreOS:n omiin Quay.io:n rekistereihin. Sille on tehty myös eri integraatioita, jotka helpottavat CoreOS Clairin käyttöä (Integrations 2017). Näistä kerrotaan seuraavissa luvuissa x-x.

6.1.2 Clair integraatio Clairctl

Clairctl on GO-ohjelmointikielellä kirjoitettu kevyt komentorivityökalu, joka tekee sillan Docker rekistereille ja CoreOS Clairille. Clairctl toimii proxyna autentikointia varten, kun yhdistetään salattuihin Docker-rekistereihin. Clairctl:lla voidaan luoda HTML-raportti Clairilta saaduista haavoittuvuuksista. (Clairctl 2016.)

Clairctl:n komentorivikomennoilla on mahdollista tarkistaa Clairin tila ja Docker imagen layerien määrä sekä niiden tarkistussummat. Layerit voidaan työntää Clairiin ja Clairctl analysoi antamalla niiden haavoittuvuuksien määrät. Myös paikallisia Docker imageja voidaan käyttää, että ei tarvitse rekisteriä käyttää. (Garcia 2016.)

6.1.3 Clair integraatio Klar

Klar on yksinkertainen työkalu Docker imageiden analysoinnissa, kun ne sijaitsevat yksityisessä tai julkisessa Docker rekisterissä. Klar hyödyntää Clairia, ja se on kirjoitettu myös GO-ohjelmointikielellä ja käyttää hyödyksi ympäristömuuttujia. Klar on yksi binaari ja ei vaadi riippuvuuksia. (Klar 2017.)

Klaria ajetaan komentoriviltä ja asetetaan optiot suoraan sitä ajettaessa kuten Clairin sijainti ja haavoittuvuuksien sallittu määrä. Klar yksinkertaisesti palauttaa komennosta numeron 0, jos asetettu määrä tiettyä haavoittuvuutta löytyy alle asetetun määrän. Numero 1 palautuu, kun haavoittuvuuksia löytyy asetetun rajan ylitse. Tätä voidaan hyödyntää CI/CD:ssä, kun komentoriville tullessa pelkkä numero 1 asettaa buildin suoraan epäonnistuneeksi. (Klar 2017.)

6.2 OWASP Dependency Check

6.2.1 Yleistä

OWASP Dependency Check on ohjelma, joka skannaa applikaation ja sen riippuvaiset kirjastot sekä tarkistaa, onko niissä tunnettuja ja julkistettuja haavoittuvuuksia. Tätä

ohjelmaa voidaan käyttää OWASP Top 10 2013 A9:n osiossa. (OWASP Dependency Check 2017.)

Pääasiassa Dependency Checker sisältää joukon analysaattoreita, jotka tarkistavat projektin riippuvuudet ja keräävät niistä tiedon palasia. Niitä käytetään identifioimaan CPE (Common Platform Enumeration) annetulle riippuvuudelle. Jos jokin CPE identifioidaan, siihen liittyvä CVE-listaus merkitään raportissa. (OWASP Dependency Check 2017.)

CPE on jäsennelty nimeämissuunnitelma informaatioteknologian järjestelmille, ohjelmistoille ja paketeille. CPE sisältää muodollisen nimiformaatin järjestelmän nimen tarkistukseen ja kuvausformaatin tekstin sekä testien yhdistämiseen nimelle. (Official Common Platform Enumeration (CPE) Dictionary 2017.)

Dependency Checkistä on komentorivin käyttöliittymä, Maven plugini, Ant tehtävä ja Jenkins plugini (OWASP Dependency Check 2017). Tästä tosin ei ole virallista työkalua Docker imagen skannaukseen ollenkaan mutta seuraavissa luvuissa 6.2.2 ja 6.2.3 esitellään kolmannen osapuolen tekemiä työkaluja.

6.2.2 Deepfenceio Dependency Checker

Deepfenceio:n motivaationa oli tuoda konttien haavoittuvuusskannaukseen portti OWASPin Dependency Checkeristä, mikä olisi samankaltainen kuin CoreOS Clair tai Atomic Scan, mutta menisi vähän pidemmälle skannaamalla riippuvuudet työkaluilla, joita on jo saatavilla ja laajasti käytössä. Deepfenceio Dependency Checker on portti OWASP Dependency Checkeristä ja Retire.js:stä Docker imageiden skannaukseen. Retire.js skannaa JavaScript ja node.js riippuvuudet, Dependency Checker skannaa muut kielet. Deepfenceio Dependency Checkeriä ajetaan suoraan kontista antaen pääsyä isäntäkoneen tiedostoihin. Lopuksi komentorivillä lukee CVE-tiedot JSON-muodossa. (Deepfenceio/deepfence_depcheck 2017.)

6.2.3 Dagda

Dagda on työkalu staattiseen analyysiin haavoittuvuuksien etsimiseen Docker imageista/konteista. Se voi myös monitoroida ajossa olevia Docker kontteja havaitakseen poikkeavia toimintoja käyttäen Sysdig Falcoa. Ensiksi CVE-tiedot, BID:it (Bugtraq ID) ja tunnetut hyväksikäyttötavat ladataan Mongo tietokantaan helpottaakseen haavoittuvuuksien ja hyväksikäyttötapojen etsinnässä, kun analyysi on käynnissä. Dagda hakee skannauksessa informaatiota ohjelmista, joita on asennettu Docker imageen kuten käyttöjärjestelmän paketit ja ohjelmointikielien riippuvuudet. Sitten se tarkistaa jokaisen tuotteen ja sen version, sisältääkö image haavoittuvuuksia, joita tallennettiin Mongo-tietokantaan. (Dagda 2017.)

Dagda tukee useaa Linux pohjaimagea: Red Hat/CentOS/Fedora, Debian/Ubuntu, OpenSUSE ja Alpine. Dagda perustuu OWASP Dependency Checkeriin ja Retire.js:iin analysoidakseen riippuvuuksia seuraavista ohjelmointikielistä: java, python, nodejs, javascript, ruby ja php. Dagdaa voidaan ajaa komentoriviltä tai REST API:n kautta. (Dagda 2017.)

7 Haavoittuvuusskannereiden asennus ja testaus

7.1 Lähtökohdat

Haavoittuvuusskannauksien toimivuutta ja ominaisuuksia testattiin VirtualBoxin virtuaalikoneella Centos 7-käyttöjärjestelmässä.

7.2 CoreOS Clair

CoreOS Clair ajetaan kokonaisuudessaan kahdessa Docker-kontissa, joista ensimmäinen on itse Clair ja toinen Postgres-tietokanta. Konttien ajamiseen on ohjeet ja Docker-imaget mainittu sen Github-sivustolla. Clair käynnistyy kontissa seuraavalla komennolla:

docker run --name clairi -d -p 6060-6061:6060-6061 -v $PWD/clair_config:/config -v /tmp:/tmp -m 1g quay.io/coreos/clair-git:latest -config=/config/config.yaml

Komento kokonaisuudessaan nimeää kontin clairiksi, asettaa käytettäväksi uusimman version Docker-imagesta, avaa siihen portit 6060 ja 6061, rajoittaa muistin käytön yhteen gigaan, asettaa Clairin asetukset volume mountilla ja määritetään käyttämään niitä. Postgres käynnistetään komennolla:

docker run -d -e POSTGRES_PASSWORD="" -p 5432:5432 -m 1g postgres:9.6

Komennossa asetetaan salasanakenttä tyhjäksi, avataan portti 5432, rajoitetaan muistin käyttö yhteen gigaan ja käytetään Docker-imagen versiota 9.6. Salasana kenttä on tyhjä testien aikana.

Tässä vaiheessa Clair hakee tunnettuja CVE-tietoja Postgres-tietokantaan. Kuviossa 12 näkyy Clairin loki käynnistyksestä, CVE-tietojen hakemisesta, CVE-tietojen lisäämisestä tietokantaan ja ilmoittaen päivittämisen valmistumisesta.

Kuvio 12. Clair käynnistysloki

Tämän jälkeen Clair on valmis vastaanottamaan Docker-imaget skannattavaksi seuraavilla kohdilla:

1. Katsotaan Docker-imagen ID, joka halutaan skannata ja tallennetaan se
2. Puretaan tallennettu tar-gz paketti, joka sisältää Docker-imagen layerit erillisissä layer.tar kansioissa ja manifest.json kertoo layereiden järjestyksen. Tar-pallot täytyy olla mountattuna Clairiin

3. Docker-imagen layereiden järjestyksen näkee Dockerin omalla historia komennolla, mikä tulee hyödyksi pohjaimagen erottamiseen
4. Lähetetään layerit Clair API:een.

Ensimmäinen kohta onnistuu katsomalla Docker-imagen ID komennolla docker images, joka tulostaa näkyville Dockerin käytettävissä olevat imaget. Dockerilla on tar-tiedostomuodoksi tallentamiseen oma komento. Esimerkkinä tallennetaan oma ubuntu image:

docker save -o 1f5283ff9b6a.tar 1f5283ff9b6a

Docker tallensi 1f5283ff9b6a.tar tiedoston, joka täytyy purkaa Clairin volume mounttiin. Aikaisemmin Clairille tehtiin volume mount /tmp kansioon, johon voidaan purkaa tar-pallo seuraavalla komennolla:

tar -xf "1f5283ff9b6a.tar" -C "/tmp/layerit/1f5283ff9b6a"

Tämä kansio sisältää Docker-imagen layerit kansioissa, jotka sisältävät layerit tiedot layer.tar tiedostossa. Linux listaus komento näyttää modifioidun ubuntu imagen layerit:

08947c9ce3a48a8b421b435754f0c5d0dc29fda77d2c4ec41e870c6262a90284
1f5283ff9b6a17b6248de464644829e5b29c1543d6f4624951c3cb0cef3d7a66.json
40624bf81f280fe80c6972ee899a217f5bd444bd6d7842cb65b39f19ce031425
7709fffefc78e2a6a806bf1046195dd6238be196c0ad3250900b5e31a1a455ca
86ea4f41569e54903bee90f048e21790dbe7e4ec21ef3b001be0909ca8bdd525
aac7491fff0c79831a27c0151c0cd1f1411f289a6797b83bc61ce0c47191da29

manifest.json

1a372565bbb07d41afe175fee5b2946f2a2b9172ec4066567be16b8b953fa215
259979e7f97312ff4bc2228db00016c83868ea21584e2649cdc3bdfedcd7e4d8
5d02f50efbc0334074d35bc14d4bf39077ee31c977ee5a78d072013c9d544a50
8025f58bac1affe153efcec14b99b456f53d0d97c60b97c78a9ed29a7e2c1fc2
93b90854cce9f6cfa7fd3002d9b3fea5253652a48f55583a472b910d1c90a648
e55369cf1b271ab28db95a8a647e09452cc5c3739417324d4e78d14672152f92

Dockerin omalla komennolla *”docker history -q --no-trunc 1f5283ff9b6a”* on tarkoitus nähdä Docker-imagen layerien järjestys. Alla on komennon tulostama tulos:

sha256:1f5283ff9b6a17b6248de464644829e5b29c1543d6f4624951c3cb0cef3d7a66

sha256:6d2e64255a95ac4336c56085fbc26fe7b55487aca554cfb6982a2d11079c6528

sha256:1fbca9414d31ee9438a89c1a4e261656163e15ac51c295282a1d3b39aa335a1f

sha256:e7966f19ffdf00c51aa23f4a1f1f7b53b149d92b82ad3117b07a36feca6c66d0

sha256:7afd513129874a2b71f60fa2edf7f22c058d2cd3b081e7299526fe1ace018268

sha256:d26e0e753e047f97125430066dc1456a74706611e23e7e61cd6ba52c551c1b35

sha256:7c4909587544968b09a44fabbce76da7848fff966717f8f4b94fcb220613c3b0

sha256:4915165cfc0ae3f89b2c7144ca686fdbf1c70ff84072a1340df0fbbbaf4dd70c

sha256:f7b3f317ec734a73deca91b34c2b1e3dd7454650da9c8ef3047d29a873865178

<missing>

<missing>

<missing>

<missing>

<missing>

Tulos osoittaa, että pohjaimagena käyttämä ubuntu ei näytä omia layereitaan vaan niiden kohdalla lukee <missing>. Ensimmäisessä rivissä on tuttu 12-merkkinen ID, joka viittaa oman Docker-imagen viimeistä layeria. Siitä alaspäin ovat aikaisemmat kerrokset. Layerien kansioiden layer.tar kertoo mitä layer pitää sisällään, mistä voi päätellä Docker-imagen layerien oikean järjestyksen. Myös manifest.json kertoo järjestyksen.

Kun järjestys on valmis, lähetetään layerit Clair API:een. Clairiin lähettäessä pitää kertoa layerin ID nimenä, polku layer.tariin, imagen formaatti ja Clairin IP-osoite. Ensimmäisen lähetyksen jälkeen lähetetään seuraava, mutta lisäksi kerrotaan edellisen layerin ID. Näin saadaan Clairille lähetettyä kokonainen Docker-image. Testissä käytetyssä imagessa oli layereita 11 ja ne lähetettiin käyttäen curl työkalua. Alla lähetetään ensimmäinen layer:

```
curl -s -H "Content-Type: application/json" -X POST -d \
'{
 "Layer": {
  "Name":
"7709fffefc78e2a6a806bf1046195dd6238be196c0ad3250900b5e31a1a455ca",
  "Path": "/tmp/lay-
erit/1f5283ff9b6a/7709fffefc78e2a6a806bf1046195dd6238be196c0ad3250900b5e3
1a1a455ca/layer.tar",
  "Format": "Docker"
 }
}' \
http://172.17.0.3:6060/v1/layers
```

Curlin vastaus tai Clairin loki kertoo, jos layer meni perille. Clairin lokissa se näyttää tältä:

```
{"Event":"Handled HTTP request","Level":"info","Loca-
tion":"router.go:57","Time":"2017-10-23 12:24:01.201994","elapsed
time":99358140,"method":"POST","remote addr":"10.118.3.78:47976","request
uri":"/v1/layers","status":"201"}
```

Seuraavien layerien lähetyksessä viitataan edelliseen layeriin *Parent Layerina*, mikä tehdään alla:

*curl -s -H "Content-Type: application/json" -X POST -d *

'{

"Layer": {

"Name": "08947c9ce3a48a8b421b435754f0c5d0dc29fda77d2c4ec41e870c6262a90284",

"Path": "/tmp/layerit/1f5283ff9b6a/08947c9ce3a48a8b421b435754f0c5d0dc29fda77d2c4ec41e870c6262a90284/layer.tar",

"Format": "Docker",

"ParentName": "7709fffefc78e2a6a806bf1046195dd6238be196c0ad3250900b5e31a1a455ca"

}

*}' *

http://172.17.0.3:6060/v1/layers

Tämä suoritetaan niin pitkään kunnes viimeinen layer on lähetetty. Alla on viimeisen layerin lähetys:

*curl -s -H "Content-Type: application/json" -X POST -d *

'{

"Layer": {

"Name": "259979e7f97312ff4bc2228db00016c83868ea21584e2649cdc3bdfedcd7e4d8",

"Path": "/tmp/layerit/1f5283ff9b6a/259979e7f97312ff4bc2228db00016c83868ea21584e2649cdc3bdfedcd7e4d8/layer.tar",

"Format": "Docker",

"ParentName": "aac7491fff0c79831a27c0151c0cd1f1411f289a6797b83bc61ce0c47191da29"

}

*}' *

http://172.17.0.3:6060/v1/layers

Nyt Clairilta voi kysyä Docker-imagen haavoittuvuudet, jotka saadaan curl työkalulla kysymällä ne viimeiseltä layeriltä. Jos haavoittuvuuksia kysytään ensimmäisiltä, ei nähdä koko Docker-imagen vakavuuksia. Viimeiseltä layerilta kysytään haavoittuvuudet komennolla:

curl -s -X GET "http://172.17.0.3:6060/v1/layers/259979e7f97312ff4bc2228db00016c83868ea21584e2649cdc3bdfedcd7e4d8?features&vulnerabilities"

Vastauksena tulee haavoittuvaiset paketit vakavuustasolla ja linkillä CVE-tietoon JSON-muodossa. Alla on esimerkkinä osa vastauksesta:

{

"Layer": {

"Name": "1a372565bbb07d41afe175fee5b2946f2a2b9172ec4066567be16b8b953fa215",

"NamespaceName": "ubuntu:16.04",

"ParentName": "e55369cf1b271ab28db95a8a647e09452cc5c3739417324d4e78d14672152f92",

"IndexedByVersion": 3,

"Features": [

{

"Name": "glibc",

"NamespaceName": "ubuntu:16.04",

"VersionFormat": "dpkg",

"Version": "2.23-0ubuntu7",

"Vulnerabilities": [

{

"Name": "CVE-2016-10228",

"NamespaceName": "ubuntu:16.04",

"Description": "The iconv program in the GNU C Library (aka glibc or libc6) 2.25 and earlier, when invoked with the -c option, enters an infinite loop when processing invalid multi-byte input sequences, leading to a denial of service.",

"Link": "http://people.ubuntu.com/~ubuntu-security/cve/CVE-2016-10228",

"Severity": "Negligible",

"Metadata": {

"NVD": {

"CVSSv2": {

"Score": 4.3,

"Vectors": "AV:N/AC:M/Au:N/C:N/I:N"

}

}

}

},

{

"Name": "CVE-2015-8985",

"NamespaceName": "ubuntu:16.04",

"Description": "The pop_fail_stack function in the GNU C Library (aka glibc or libc6) allows context-dependent attackers to cause a denial of service (assertion failure and application crash) via vectors related to extended regular expression processing.",

"Link": "http://people.ubuntu.com/~ubuntu-security/cve/CVE-2015-8985",

"Severity": "Low",

"Metadata": {

"NVD": {

"CVSSv2": {

"Score": 4.3,

"Vectors": "AV:N/AC:M/Au:N/C:N/I:N"

}

}

}

},

{

"Name": "CVE-2015-5180",

"NamespaceName": "ubuntu:16.04",

"Description": "DNS resolver NULL pointer dereference with crafted record type",

"Link": "http://people.ubuntu.com/~ubuntu-security/cve/CVE-2015-5180",

"Severity": "Low"

},

{

"Name": "CVE-2017-8804",

"NamespaceName": "ubuntu:16.04",

"Description": "The xdr_bytes and xdr_string functions in the GNU C Library (aka glibc or libc6) 2.25 mishandle failures of buffer deserialization, which allows remote attackers to cause a denial of service (virtual memory allocation, or memory consumption if an overcommit setting is not used) via a crafted UDP packet to port 111, a related issue to CVE-2017-8779.",

"Link": "http://people.ubuntu.com/~ubuntu-security/cve/CVE-2017-8804",

"Severity": "Medium"

}

],

"AddedBy": "7709fffefc78e2a6a806bf1046195dd6238be196c0ad3250900b5e31a1a455ca"

}

7.3 Clair integraatio Clairctl

Clairctl varten asennettiin Go-kääntäjä. Sen asennuspaketti haettiin curlilla väliaikaiseen polkuun /tmp ja purettiin tarrapallosta pois polkuun /usr/local komennoilla:

```
cd /tmp

curl -LO https://storage.googleapis.com/golang/go1.7.linux-amd64.tar.gz

sudo tar -C /usr/local -xvzf go1.7.linux-amd64.tar.gz
```

Go-kääntäjälle luotiin työskentelykansioita kotihakemistoon, johon haetaan myöhemmin Clairctl Go-koodit:

```
mkdir -p ~/projects/{bin,pkg,src}
```

Jotta Go:ta voidaan ajaa polusta */usr/local/go/bin*, se täytyy lisätä $PATH:iin ympäristömuuttujana globaaliseksi. Tiedostoon */etc/profile.d/path.sh* lisätään seuraava rivi:

export PATH=$PATH:/usr/local/go/bin

Lisätään työskentelykansiot myös ympäristömuuttujiksi, että Go osaa kertoa Go-koodien paikat toimiakseen. Nämä lisätään käyttäjän *.bash_profile* tiedostoon:

export GOBIN="$HOME/projects/bin"

export GOPATH="$HOME/projects/src"

Nämä exportit otetaan käyttöön ajamalla komento:

source /etc/profile && source ~/.bash_profile

Clairctl vaatii Gliden, joka asennetaan komennolla:

curl https://glide.sh/get | sh

Nyt Clairctl Go-koodi voidaan hakea *git clonella*, asentaa sen tarvittavat riippuvuuden ja rakentaa Clairctl binaari:

git clone github.com/jgsqware/clairctl $GOPATH/src/github.com/jgsqware/clairctl

cd $GOPATH/src/github.com/jgsqware/clairctl

glide install -v

go get -u github.com/jteeuwen/go-bindata/...

go generate ./clair

go build

Clairctl binaari rakentuu polkuun *$GOPATH/src/github.com/jgsqware/clairctl/clairctl*. Clairctl toimivuus testattiin sen healthcheckillä, joka testaa yhteyden Clairiin. Tässä komennossa käytettiin aiemmin luotua Clair-konttia ja tuloksena tulee oikein tai väärin merkki:

sudo $GOPATH/src/github.com/jgsqware/clairctl/clairctl health

Clair: ✓

Clairctl sisältää useita komentoja. Clairctl pull hakee Docker imagen layerit, mikä on hyvä tarkistus ennen niiden lähettämistä Clairiin. Alla käytetään lokaalina löytyvää Docker imagea ja samalla tulostetaan enemmän lokia vianselvennystä varten:

sudo $GOPATH/src/github.com/jgsqware/clairctl/clairctl --log-level debug pull --local ubuntu/mongotesti:1

Clairctl push lähettää imagen layerit Clairille:

sudo $GOPATH/src/github.com/jgsqware/clairctl/clairctl --log-level debug push --local ubuntu/mongotesti:1

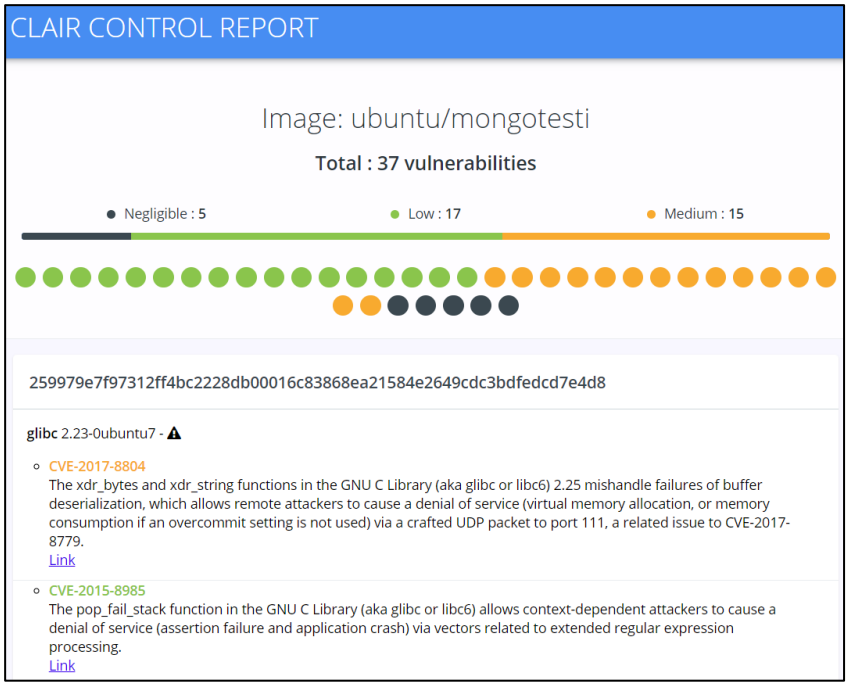
Clairctl analyze lähettää imagen layerit Clairille ja kertoo layerin haavoittuvuudet:

sudo $GOPATH/src/github.com/jgsqware/clairctl/clairctl --log-level debug analyze --local ubuntu/mongotesti:1

Clairctl report luo HTML-raportin löydetyistä tietoturvahaavoittuvuuksista:

sudo $GOPATH/src/github.com/jgsqware/clairctl/clairctl --log-level debug report --format html --local ubuntu/mongotesti:1

Kuviossa 13 on luotu HTML-raportti, josta löytyy haavoittuvaiset paketit, niiden vakavuudet ja linkit CVE-tietoihin.



Kuvio 13. Clairctl HTML-raportti haavoittuvuuksista

7.4 Clair integraatio Klar

Klar vaatii myös Go-kääntäjän, mutta pelkkä Klarin koodin hakeminen ilman asennusta riittää, että ei tarvitse asentaa sitä enää. Kun Klar oli haettu komennolla "go get github.com/optiopay/klar" Go:n binaarikansioon, sitä voitiin käyttää suoraan Linuxin komentoriviltä. Työkalua testattiin komennolla:

CLAIR_ADDR=http://clair /home/jenkins/projects/bin/klar ubuntu:latest

Komennossa määritettiin konttissa ajettavan Clairin IP-osoite ja skannattava Docker image. Vastauksena tuli "Can't pull image: Token request returned 401." Tämä johtuu siitä, että Dockerilla täytyy olla kirjauduttuna Docker rekisteriin, josta image haetaan ja skannataan. Vaikka Docker image löytyy Dockerilla lokaalisti, sama virhe toistuu. Numero 401 on http error 401, joka tarkoittaa luvatonta käyttöä, koska kirjautumista

ei tehty. Vastauksena pitäisi tulla "Found x vulnerabilities”, jossa x olisi löydettyjen haavoittuvuuksien määrä.

7.5 Deepfenceio Dependency Checker

Deepfenceio:n Dependency Checker on Docker-kontista ajettava sovellus, joten sitä ajetaan docker run komennolla mukaan lukien tietokannan luonti haavoittuvuuksista ja Docker imageiden skannaukset. Tietokanta luotiin virtuaalikoneelle kansioon, joka oli mountattuna Docker-konttiin. Luonnin aikana Dockerin socketti, kirjastopolku ja rootti piti myös mountata ohjeiden mukaan. Tietokanta luotiin alla olevalla komennolla, jossa käytettiin optiona -u true. Komennon alla näkyy kontin tulostama teksti:

docker run -ti -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/docker/:/fenced/mnt/host/var/lib/docker/:rw -v /:/fenced/mnt/host/:ro -v /home/vc/db:/tmp:rw deepfenceio/deepfence_depcheck -u true

 # --

 # Deepfence Dependency Check (OWASP Dependency Checker + Retire.js) for Container Images

 # Running with following config

#

 # Container image = host

 # Scan type = all

 # Proxy = none

 # Debug messages logged to = stdout

 # Vulnerabilities logged to = stderr, /tmp/depcheck/host

 # DB update = true

 # Databases stored at = /tmp/dependency-check, /tmp/.retire-cache

 # JSON pretty printing = false

 # --

[INFO] OWASP Dependency Check is building initial database

[INFO] Retirejs is building initial database

[INFO] Done

Tietokannan luonti kesti noin seitsemän minuuttia, minkä jälkeen voi skannata Docker imageita. Optiolla -i node kerrottiin Docker imagen nimi, missä esimerkkinä on node image. Alla on sama komento uudella optiolla ja sen antama tulos:

docker run -ti -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/docker/:/fenced/mnt/host/var/lib/docker/:rw -v /:/fenced/mnt/host/:ro -v /home/vc/db:/tmp:rw deepfenceio/deepfence_depcheck -i node

 # --

 # Deepfence Dependency Check (OWASP Dependency Checker + Retire.js) for Container Images

 # Running with following config

#

 # Container image = node

 # Scan type = all

 # Proxy = none

 # Debug messages logged to = stdout

 # Vulnerabilities logged to = stderr, /tmp/depcheck/node

 # DB update = false

 # Databases stored at = /tmp/dependency-check, /tmp/.retire-cache

 # JSON pretty printing = false

 # --

[INFO] Saving node

[INFO] Getting image history

[INFO] Image node has 8 layers which need scanning

{"cve_id":"bug-id-11290","cve_type":"js","cve_container_image":"node","cve_severity":"medium","cve_caused_by_package":"jquery-1.7.2","cve_container_layer":"2c1f40d031f6c219d137d15aca69d5b0f456fbf9e91412c314d73e333052c7a0","cve_fixed_in":"Unknown","cve_link":"[http://bugs.jquery.com/ticket/11290 http://research.insecurelabs.org/jquery/test/]","cve_description":"Selector interpreted as HTML","cve_cvss_score":"0.00","cve_attack_vector":"Unknown"}

{"cve_id":"git-issue-2432","cve_type":"js","cve_container_image":"node","cve_severity":"medium","cve_caused_by_package":"jquery-1.7.2","cve_container_layer":"2c1f40d031f6c219d137d15aca69d5b0f456fbf9e91412c314d73e333052c7a0","cve_fixed_in":"Unknown","cve_link":"[https://github.com/jquery/jquery/issues/2432 http://blog.jquery.com/2016/01/08/jquery-2-2-and-1-12-released/]","cve_description":"3rd party CORS request may execute","cve_cvss_score":"0.00","cve_attack_vector":"Unknown"}

{"cve_id":"bug-id-11290","cve_type":"js","cve_container_image":"node","cve_severity":"medium","cve_caused_by_package":"jquery-1.7.2","cve_container_layer":"2c1f40d031f6c219d137d15aca69d5b0f456fbf9e91412c314d73e333052c7a0","cve_fixed_in":"Unknown","cve_link":"[http://bugs.jquery.com/ticket/11290 http://research.insecurelabs.org/jquery/test/]","cve_description":"Selector interpreted as HTML","cve_cvss_score":"0.00","cve_attack_vector":"Unknown"}

{"cve_id":"git-issue-2432","cve_type":"js","cve_container_image":"node","cve_severity":"medium","cve_caused_by_package":"jquery-1.7.2","cve_container_layer":"2c1f40d031f6c219d137d15aca69d5b0f456fbf9e91412c314d73e333052c7a0","cve_fixed_in":"Unknown","cve_link":"[https://github.com/jquery/jquery/issues/2432 http://blog.jquery.com/2016/01/08/jquery-2-2-and-1-12-released/]","cve_description":"3rd party CORS request may execute","cve_cvss_score":"0.00","cve_attack_vector":"Unknown"}

Komento palautti kaksi haavoittuvuustietoa duplikaattina Docker imagesta node.

7.6 Dagda

Dagda koostuu Mongo-tietokannasta ja Dagda-python sovelluksesta. Mongo-tietokanta laitettiin Docker-konttiin pyörimään portilla 27017 komennolla:

docker run -d -p 27017:27017 mongo

Dagda vaatii vähintään pythonin version 3.3.X ja Pip3:sen sekä siihen usean suosituksen. Nämä asentuvat komennoilla:

sudo yum -y install https://centos7.iuscommunity.org/ius-release.rpm

sudo yum -y install python36u

sudo yum -y install python36u-pip

sudo pip3.6 install -r requirements.txt

Requirements.txt sisältää Pip3:seen PyMongo, Requests, Python-dateutil, Joblib, Docker, , Flask, Flask-cors, PyYAML ja Defusedxml. Dagda vaatii myös kernel headerit virtuaalikoneen käyttöjärjestelmältä Sysdig Falcoa varten. Nämä asennettiin komennoilla:

sudo yum -y install kernel-devel-$(uname -r)

sudo yum -y install yum-utils

sudo yum -y groupinstall development

sudo yum -y install dkms

Seuraavaa komento ajettiin, että ei tule virheilmoitusta Sysdig Falcosta liittyen sysdig_probe moduuliin:

sudo /usr/lib/dkms/dkms_autoinstaller start

Bash-profiiliin lisättiin vielä pari ympäristömuuttujaa Dagdaa varten, millä kerrotaan Dagdan sijainti ja käytettävä portti. Komennolla *"vi ~/.bash_profile"* laitettiin *"export DAGDA_HOST='127.0.0.1'"* ja *"export DAGDA_PORT=5000"*. Nämä otettiin käyttöön komennolla:

source ~/.bash_profile

Nyt Dagdan voi käynnistää komennolla, jossa Mongodb:n IP-osoite tulee Docker-kontista ja se ilmoitettiin Dagdaa varten:

python3.6 /home/vc/dagda/dagda/dagda.py start --mongodb_host 172.17.0.2

CVE-tiedot ladataan tietokantaan komennolla, jota käytetään myös sen päivittämiseen uusien CVE-tietojen varalta. Tämä kestää useita minuutteja:

python3.6 /home/vc/dagda/dagda/dagda.py vuln --init

Haluttu Docker image skannataan seuraavalla komennolla, jonka alla sen antama tulos:

python3.6 /home/vc/dagda/dagda/dagda.py check --docker_image ubuntu/mongotesti:1

{

"id": "5922830ee1382309bc0a75a5",

"msg": "Accepted the analysis of <ubuntu/mongotesti:1>"

}

Tulos kertoo, että Dagda on vastaanottanut Docker imagen skannattavaksi. Skannaus kestää useita minuutteja. Jos skannaus ei ole valmis, Dagda kertoo siitä. Haavoittuvuustiedot saadaan selville kysymällä edellisen tuloksen ID:llä Dagdalta komennolla:

python3.6 /home/vc/dagda/dagda/dagda.py history ubuntu/mongotesti:1 --id 5922830ee1382309bc0a75a5

Tuloksena tulee JSON-muodossa 1703 riviä tekstiä Docker imagen käyttöjärjestelmäpaketeista, niiden versioista ja tieto, että sisältääkö ne haavoittuvuuksia. Alla on esimerkkinä ensimmäiset 62 riviä, joista 32 riviä kertoo haavoittumattomat paketit ja loput haavoittuvuuden sisältävän bashin:


```
[
  {
    "id": "5922830ee1382309bc0a75a5",
    "image_name": "ubuntu/mongotesti:1",
    "static_analysis": {
      "os_packages": {
        "ok_os_packages": 150,
        "os_packages_details": [
          {
            "is_vulnerable": false,
            "product": "adduser",
            "version": "3.113+nmu3ubuntu4",
            "vulnerabilities": []
          },
          {
            "is_vulnerable": false,
```

```
    "product": "apt",
    "version": "1.2.19",
    "vulnerabilities": []
},
{
    "is_vulnerable": false,
    "product": "base-files",
    "version": "9.4ubuntu4.4",
    "vulnerabilities": []
},
{
    "is_vulnerable": false,
    "product": "base-passwd",
    "version": "3.5.39",
    "vulnerabilities": []
},
{
    "is_vulnerable": true,
    "product": "bash",
    "version": "4.3",
    "vulnerabilities": [
      {
          "CVE-2014-6271": {
              "cveid": "CVE-2014-6271",
```

"cvss_access_complexity": "Low",

"cvss_access_vector": "Network",

"cvss_authentication": "None required",

"cvss_availability_impact": "Complete",

"cvss_base": 10.0,

"cvss_confidentiality_impact": "Complete",

"cvss_exploit": 10.0,

"cvss_impact": 10.0,

"cvss_integrity_impact": "Complete",

"cvss_vector": [

"AV:N",

"AC:L",

"Au:N",

"C:C",

"I:C",

"A:C"

],

"cweid": "CWE-78",

"mod_date": "06-01-2017",

"pub_date": "24-09-2014",

"summary": "GNU Bash through 4.3 processes trailing strings after function definitions in the values of environment variables, which allows remote attackers to execute arbitrary code via a crafted environment, as demonstrated by vectors involving the ForceCommand feature in OpenSSH sshd, the mod_cgi and mod_cgid modules in the Apache HTTP Server, scripts executed by unspecified DHCP

clients, and other situations in which setting the environment occurs across a privilege boundary from Bash execution, aka \"ShellShock.\" NOTE: the original fix for this issue was incorrect; CVE-2014-7169 has been assigned to cover the vulnerability that is still present after the in correct fix."

}

8 Haavoittuvuusskannereiden vertailu

8.1 Kvalitatiivinen tutkimus

Kvalitatiivisen eli laadullisen tutkimuksen tarkoituksena on perehtyä kohteeseen tai kohteisiin ottaen huomioon mahdollisimman paljon havaintoja, joita hyödynnetään lopputuloksessa. Tärkeää on pysyä lähtökohdissa, kun tutkittavia kohteita aletaan ymmärtämään, sillä ne voivat muovautua tutkimuksen aikana. Tutkimuksessa ei todenneta olemassa olevia totuuksia vaan pyritään löytämään tosiasioita. (Hirsjärvi, Remes & Sajavaara 2007, 157).

Kvalitatiivista tutkimusta tehtiin muutaman syyn takia. Haavoittuvuusskannereiden vertailussa perehdyttiin jokaiseen tuotteeseen erikseen, että tosiasioiden ja havaintojen löytämistä suoritettiin tasapuolisesti. Lähtökohtana oli löytää potentiaaliset haavoittuvuusskannerit, joita otettiin käyttöön tuotannossa tai huomioitiin mahdollisiin tuleviin projekteihin. Lähtökohtaa huomioitiin ja mietittiin koko vertailun aikana.

8.2 Vertailu ja tutkimus

Ensisilmäyksellä CoreOS Clairilla näytti olevan tulevaisuutta ja tukea, kun se on käytössä quay.io repositoryissakin. Avoimen lähdekoodin ansiosta siitä voidaan ilmoittaa bugeista ja valmiista korjauksista githubin kautta. CoreOS Clairin pystyttäminen oli hyvin yksinkertaista: kaksi Docker-konttia yhteyksineen ja automaattinen tietokannan täyttyminen CVE-tietoineen riittää. Konfiguraatiotiedostoon voidaan asettaa, kuinka usein tietokanta päivitetään. Konttien hajoaminen ei myöskään haittaa, koska

tietokanta täytetään tiedoilla uudelleen. Ainoat miinukset olivat haavoittuvuuksien saaminen pelkistä käyttöjärjestelmän paketeista ja sen monimutkaisesta käyttämisestä. Ohjelmien kirjastojen skannaus olisi ollut plussaa. Tuloksien saaminen yksittäiseltä Docker imagelta vaati paljon manuaalista tutkimista ja käsin tehtäviä komentoja. Testiympäristössä ubuntun haavoittuvuuksien hakeminen bzr:llä oli vienyt todella paljon muistia ja sen seurauksena välillä jättänyt niiden hakemisen.

CoreOS Clairin avuksi integraatio oli manuaalisen käytön takia välttämätön. Näistä löytyi Klar ja Clairctl github kautta. Klar oli tosi yksinkertainen ja asennus helppoa. Binaarin luominen oli mahdollista, jos sen lisäisi kevyenä skannerina Docker imageen. Se kertoo haavoittuvuuksien määrän suoraan tuloksena ja haluttaessa haavoittuvuuksien tiedot. Se osaa suoraan palauttaa numeron 1, kun on haavoittuvuuksia liikaa. Klarin heikkoutena on lokaalin Docker imagen skannauksen uupuminen. CI/CD-prosessissa Docker image täytyy työntää Docker-rekisteriin Docker imagen rakennuksen jälkeen ja sitten vasta tehdä skannaus. Tässä turhaan työnnetään Docker image rekisteriin, jos skannauksessa ilmenee haavoittuvuuksia. Julkisesta Docker-rekisteristäkään se ei halunnut hakea Docker imagea ja skannata sitä. Testissä Docker rekisteriin ei ollut tunnuksia ja dokumentaation perusteella siitä ei saanut muuta irti.

Clairctl integraation asennus oli myös helppoa ja siitä saisi binaarin luotua. Clairctl sisälsi paljon ominaisuuksia kuten Docker imagen layerien viennin ja analyysin tekemisen yhdellä komennolla. HTML-raportin luominen oli iso plussa, koska sen voi liittää CI/CD-prosessiin. Siitä on mahdollista katsoa helposti mitä haavoittuvuuksia löytyy ja suorat linkit CVE-tietoihin. Raportin luonnissa välillä tuli ongelmia muistin loppumisesta.

Deepfenceio Dependency Checkeristä oli oma Docker image valmiina, mutta se oli pelkkänään noin 300 MB. Skannaus kesti paljon pitempään verrattuna Clairctl:n aikaan. Isojen Docker imageiden skannauksessa voi esiintyä ongelmia, mitä ei haluta CI/CD-prosessia keskeyttämään. Sen ohjelmistokirjastojen skannaus oli hyvä ominaisuus. Yleistä tietokantaa ei käytetty, mutta sen olemassaolo tuskin haittaa, kun tallennetaan CVE-tiedot levylle tiedostoina. Sen päivittämisen voisi suorittaa cronin avulla. Docker imagesta on olemassa vain latest-versio, joten paluu edellisiin versioihin on mahdotonta.

Dagda tulostaa yksityiskohtaisen määrän skannaustuloksestaan, mutta kertoo myös haavoittumattomat ohjelmistopaketit. Dagdan skannaustuloksen saaminen kesti useita minuutteja ja se piti kysyä manuaalisesti omalla komennolla. CI/CD-prosessissa tämä on huono, kun ei tiedä milloin se voi jatkaa prosessiaan. Asentaminenkin vaati hieman liikaa työtä, mutta muuten Dagda oli työssään mainio.

Lopputuloksena valittiin CoreOS Clair ja siihen avuksi Clairctl CI/CD-prosessiin. CoreOS Clair on kuulu ja sille löytyy tukea jatkossakin. Sen selkeä pystytys ja konfigurointi sopivat Kubernetekseen laitettavaksi. Integraatio Clairctl otettiin avuksi sen yksinkertaisuudestaan ja hyvistä ominaisuuksista. HTML-raportin luominen oli näppärä, koska siitä näkee selkeästi mitä haavoittuvuuksia löytyy ja kuinka vakavia ne ovat. OWASP Dependency Check (Dagda) huomioidaan jatkokehityksessä. Sen voisi ajastaa skannaamaan Docker-rekisterin tai käynnissä olevien konttien ohjelmistopakettien haavoittuvuuksia. Toimeksiantaja oli samaa mieltä valitusta haavoittuvuusskannerista.

9 Kuberneteksen tietoturva

9.1 Lähtökohdat

Käytössä on Kubernetes-ympäristö Googlen Container Enginellä, jossa huomioitiin erityisesti moniasiakasympäristöjä ja sovellustuotannon näkökulmaa. Testejä suoritettiin erillisellä kahden worker noden klusterilla, että haittaa ei ilmene tuotannon konttien toiminnassa. Kubernetes klusterin versio oli 1.7.6. Testien tuloksia hyödynnetään parantamaan tietoturvaa tuotannon konttiympäristössä.

9.2 ResourceQuotas

ResourceQuota objektilla voidaan rajata ja luvata resurssien käyttöä. Resursseina voi olla esimerkkinä prosessori, muisti ja objektien lukumäärä. Objektissa voidaan määrittää resurssien rajat ja niiden ylittyessä objektien luominen estetään. Kuviossa 14

on määritetty ResourceQuota objekti nimellä compute-resources, jossa podien rajaksi on asetettu kaksi kappaletta namespacessa testispace3.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
  namespace: testispace3
spec:
  hard:
    pods: "2"
```

Kuvio 14. ResourceQuota YAML-tiedosto

Kuviossa 15 otetaan käyttöön edellisen kuvion ResourceQuotas objekti ja luodaan uusia podeja deploymentilla namespaceen testispace3. Kun kaksi podia on jo käytössä, kolmatta ei tule näkyviin "kubectl get pods" komennolla, vaikka deployment on luotu.

```
kuoppala_eerik@custom-producer-181711:~$ kubectl apply -f resourcequota.yaml
resourcequota "compute-resources" created
kuoppala_eerik@custom-producer-181711:~$ kubectl describe quota compute-resources -n testispace3
Name:           compute-resources
Namespace:      testispace3
Resource        Used    Hard
--------        ----    ----
pods            1       2
kuoppala_eerik@custom-producer-181711:~$ kubectl run nginx-2 --image=nginx:latest -n testispace3
deployment "nginx-2" created
kuoppala_eerik@custom-producer-181711:~$ kubectl describe quota compute-resources -n testispace3
Name:           compute-resources
Namespace:      testispace3
Resource        Used    Hard
--------        ----    ----
pods            2       2
kuoppala_eerik@custom-producer-181711:~$ kubectl get pods -n testispace3
NAME                       READY     STATUS    RESTARTS   AGE
nginx-2-1179057700-l1nlr   1/1       Running   0          11s
nginx-726417742-0n1bz      1/1       Running   0          19h
kuoppala_eerik@custom-producer-181711:~$ kubectl run nginx-3 --image=nginx:latest -n testispace3
deployment "nginx-3" created
kuoppala_eerik@custom-producer-181711:~$ kubectl get pods -n testispace3
NAME                       READY     STATUS    RESTARTS   AGE
nginx-2-1179057700-l1nlr   1/1       Running   0          1m
nginx-726417742-0n1bz      1/1       Running   0          19h
kuoppala_eerik@custom-producer-181711:~$ kubectl describe quota compute-resources -n testispace3
Name:           compute-resources
Namespace:      testispace3
Resource        Used    Hard
--------        ----    ----
pods            2       2
kuoppala_eerik@custom-producer-181711:~$
```

Kuvio 15. ResourceQuotan käyttöönotto ja testaus

Kuviossa 16 nähdään, että deployment ei luo podia, koska ResourceQuotas estää sen.

```
kuoppala_eerik@custom-producer-181711:~$ kubectl get deploy -n testispace3
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     1         1         1            1           19h
nginx-2   1         1         1            1           6m
nginx-3   1         0         0            0           6m
```

Kuvio 16. ResourceQuota estää deploymentin podin

Kuviossa 17 nähdään, että luodessa podia YAML-tiedostosta testispace3 namespaceen, kun podeja on jo kaksi kappaletta namespacessa testispace3, se estetään myös.

```
kuoppala_eerik@custom-producer-181711:~$ kubectl apply -f pod.yaml
Error from server (Forbidden): error when creating "pod.yaml":
 pods "nginx-pod" is forbidden: exceeded quota: compute-resources, requested: pods=1, used: pods=2, limited: pods=2
```

Kuvio 17. ResourceQuota estää podin luonnin YAML-tiedostosta

Resurssien rajoituksia on mahdollista laittaa suoraan podin objektiin estäen liiallisen resurssien käytön Kubernetes worker nodelta. Kuviossa 18 luodaan podi, joka voi käyttää maksimissaan worker nodella puolikkaan prosessorin tehot. Docker imagena käytetään vish/stressiä, jossa voidaan määrittää käynnistysoptioissa prosessorin kuorma testejä varten. Testiksi annettiin käytettäväksi kokonainen prosessori.

```
apiVersion: v1
kind: Pod
metadata:
name: cpu-demo
spec:
containers:
- name: cpu-demo-ctr
    image: vish/stress
    resources:
    limits:
        cpu: 500m
    requests:
        cpu: 100m
    args:
    - -cpus
    - "1"
```

Kuvio 18. Resurssien asettaminen podin YAML-tiedostoon

Kuviossa 19 katsotaan topilla kuorman käyttöä ja voidaan huomata rajauksen toimivan, koska käyttö ei mene luvattuun 1000m. Ensimmäisen worker noden topilla huomataan käytön nousseen 568m ja toisella worker nodella käyttö on pysynyt samana 56m.

```
kubectl top pod cpu-demo
NAME       CPU(cores)   MEMORY(bytes)
cpu-demo   501m         0Mi

kubectl top node gke-cluster-1-default-pool-f1e03554-421c
NAME                                        CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
gke-cluster-1-default-pool-f1e03554-421c   568m         60%       2070Mi          78%

kubectl top node gke-cluster-1-default-pool-f1e03554-rkz4
NAME                                        CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
gke-cluster-1-default-pool-f1e03554-rkz4   56m          5%        1802Mi          68%
```

Kuvio 19. Kuorman käytön tarkistus rajauksen jälkeen

Testattu resurssien rajaus konteilta havaittiin tärkeäksi, koska liiallisten resurssien käyttäminen virtuaalikoneelta heikentää muiden palveluiden toimintaa. Jos kontin kuorma nousee odottamattomasta syystä, sen rajaus estää liiallisen kuorman käytön ja virheiden tapahtumat. Toimeksiantajalle tämä on tärkeää, kun asiakkaiden palveluita ajetaan tuotannossa, että kummallekaan osapuolelle ei aiheudu haittaa. Testin perusteella resurssien takaamista tullaan myös hyödyntämään.

9.3 NetworkPolicy

NetworkPolicy objektilla voidaan rajata podien pääsyjä Kubernetes ympäristössä. Oletuksena podit voivat liikennöidä toisiinsa, vaikka ne olisivat eri namespaceissa. Kubernetes versiossa 1.7.6 on betassa Calico, joka on NetworkPolicy kontrolleri. Tämä vaatii 2 kpl n1-standard-1 nodea ja suositellaan kolmen käyttöä. Ilman Calicoa tai muuta kontrolleria, NetworkPolicyn luonnilla ei ole vaikutusta. Kuviossa 20 on listattuna Kuberneteksen hallintaan tarkoitetut podit ennen Calicon asennusta yhdellä nodella.

```
kubectl get pods -n kube-system
NAME                                                   READY     STATUS    RESTARTS   AGE
event-exporter-1421584133-xg478                        2/2       Running   0          1h
fluentd-gcp-v2.0-81pqd                                 2/2       Running   0          1h
heapster-v1.4.2-1873998994-xkdtg                       2/2       Running   0          1h
kube-dns-3092422022-ds5zp                              3/3       Running   0          1h
kube-dns-autoscaler-97162954-0916b                     1/1       Running   0          1h
kube-proxy-gke-cluster-1-default-pool-f1e03554-rkz4    1/1       Running   0          1h
kubernetes-dashboard-2485788342-955j5                  1/1       Running   0          1h
l7-default-backend-1798834265-0hw8t                    1/1       Running   0          1h
```

Kuvio 20. Podit yhdellä nodella ennen Calicon asennusta

Calico saadaan käyttöön Google Cloud Consolesta ja komentoriviä käyttäen. Alla on kaksi komentoa, jotka asettavat NetworkPolicyn käyttöön. Komennoissa optioina

käytetään Google projektia custom-producer-181711, klusteria cluster-1 ja vyöhykettä europe-west3-a:

gcloud beta container clusters update cluster-1 --project=custom-producer-181711 --zone=europe-west3-a --update-addons=NetworkPolicy=ENABLED

gcloud beta container clusters update cluster-1 --project=custom-producer-181711 --zone=europe-west3-a --enable-network-policy

Kun komennot ovat asetettu ja odotettu niiden astumista voimaan, Calicosta tulee muutama podi käyttöön, mutta ne jäävät Pending-tilaan, kuten kuviossa 21 nähdään.

```
kubectl get pods -n kube-system
NAME                                                  READY     STATUS    RESTARTS   AGE
calico-node-29g45                                     2/2       Running   0          8m
calico-node-vertical-autoscaler-4066868540-3cs8q      0/1       Pending   0          11m
calico-typha-650841560-nbxn7                          1/1       Running   0          11m
calico-typha-horizontal-autoscaler-2221501996-nj42h   0/1       Pending   0          11m
calico-typha-vertical-autoscaler-2910497665-1sjtk     0/1       Pending   0          11m
event-exporter-1421584133-16wnn                       0/2       Pending   0          11m
fluentd-gcp-v2.0-hp0dk                                2/2       Running   0          8m
heapster-v1.4.2-1873998994-xbr1g                      2/2       Running   0          11m
ip-masq-agent-q6wsh                                   1/1       Running   0          8m
kube-dns-3092422022-81bpg                             3/3       Running   0          11m
kube-dns-autoscaler-97162954-2wt3r                    1/1       Running   0          11m
kube-proxy-gke-cluster-1-default-pool-f1e03554-rkz4   1/1       Running   0          8m
kubernetes-dashboard-2485788342-xtnql                 0/1       Pending   0          11m
l7-default-backend-1798834265-fgmv2                   0/1       Pending   0          11m
```

Kuvio 21. Podit yhdellä nodella Calicon asennuksen jälkeen

Kun worker noden määrää nostetaan kahteen, kuviossa 22 huomataan kaikkien podien olevan päällä ja muutaman tuplaantuneen.

```
kubectl get pods -n kube-system
NAME                                                   READY     STATUS    RESTARTS   AGE
calico-node-29g45                                      2/2       Running   0          56m
calico-node-nxfvc                                      2/2       Running   0          5m
calico-node-vertical-autoscaler-4066868540-3cs8q       1/1       Running   0          59m
calico-typha-650841560-nbxn7                           1/1       Running   0          59m
calico-typha-horizontal-autoscaler-2221501996-nj42h    1/1       Running   0          59m
calico-typha-vertical-autoscaler-2910497665-lsjtk      1/1       Running   0          59m
event-exporter-1421584133-16wnn                        2/2       Running   0          59m
fluentd-gcp-v2.0-hp0dk                                 2/2       Running   0          56m
fluentd-gcp-v2.0-nzq0f                                 2/2       Running   0          5m
heapster-v1.4.2-1873998994-xbr1g                       2/2       Running   0          59m
ip-masq-agent-9hznc                                    1/1       Running   0          5m
ip-masq-agent-q6wsh                                    1/1       Running   0          56m
kube-dns-3092422022-7ng11                              3/3       Running   0          4m
kube-dns-3092422022-81bpg                              3/3       Running   0          59m
kube-dns-autoscaler-97162954-2wt3r                     1/1       Running   0          59m
kube-proxy-gke-cluster-1-default-pool-f1e03554-421c    1/1       Running   0          4m
kube-proxy-gke-cluster-1-default-pool-f1e03554-rkz4    1/1       Running   0          56m
kubernetes-dashboard-2485788342-xtnql                  1/1       Running   0          59m
l7-default-backend-1798834265-fgmv2                    1/1       Running   0          59m
```

Kuvio 22. Podit kahdella nodella Calicon asennuksen jälkeen

Kuviossa 23 on määritelty NetworkPolicy-objekti, jolla testataan samassa testispace1-namespacessa olevan Wordpress-podin pääsyä MySQL-podiin. Samalla objektilla annetaan kaikille podeille pääsy toisesta namespacesta. Objektin alussa specin alla podSelector ja matchLabels määrittävät, mihin podiin rajataan pääsyt. Rajaus koskee podia, jolla on label app:mysql. Ingressin alle kootaan podit ja namespacet, jotka voivat liikennöidä määritettyyn MySQL-podiin. Wordpress-podin liikennöinti sallitaan samalla tavalla käyttäen podSelectoria ja matchLabelsia kuin äskenkin. Wordpress-podissa on labelina app:wordpress, jolloin se voi liikennöidä. Namespacelle sallitaan pääsyt namespaceSelectorilla, johon asetetaan myös matchLabels. Namespace, jonka label on test:space, voi liikennöidä MySQL-podille.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: mysql-policy
  namespace: testispace1
spec:
  podSelector:
    matchLabels:
      app: mysql
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          test: space
    - podSelector:
        matchLabels:
          app: wordpress
    ports:
    - protocol: TCP
      port: 3306
```

Kuvio 23. NetworkPolicy YAML-tiedosto

Kuviossa 24 kokeillaan NetworkPolicyn toimivuutta. Siinä ajetaan MySQL-konttia labelilla app:wordpress ja namespacessa testispace1, jolloin liikennöinti tietokantaan onnistuu. Lopuksi yritetään katsoa tietokannan sisältöä.

```
kubectl run -it --rm --image=mysql:5.6 --labels="app=wordpress" \
-n testispace1 --restart=Never mysql-client -- mysql --host=mysql --password=Kissa123! --database=wordpress

mysql> show tables;
+-----------------------+
| Tables_in_wordpress   |
+-----------------------+
| wp_commentmeta        |
| wp_comments           |
| wp_links              |
| wp_options            |
| wp_postmeta           |
| wp_posts              |
| wp_term_relationships |
| wp_term_taxonomy      |
| wp_termmeta           |
| wp_terms              |
| wp_usermeta           |
| wp_users              |
+-----------------------+
12 rows in set (0.00 sec)
```

Kuvio 24. NetworkPolicy testi

Tietokantaan pääsee toisesta namespacesta kiinni, kun sen labelina on test:space. Alla olevalla komennolla onnistuu testaus, missä hostnamea muokataan toimivaksi Kuberneteksen DNS-nimien toiminnan vuoksi:

kubectl run -it --rm --image=mysql:5.6 -n testispace2 --restart=Never mysql-client -- mysql --host=mysql.testispace1 --password=Kissa123! --database=wordpress

Mistään muusta podista tai namespacesta ei voi liikennöidä MySQL-podiin, jos labelit eivät ole NetworkPolicyn mukaiset.

Testattu NetworkPolicyn Calico on tärkeä, koska sillä voi suojata podiin tulevat turhat ja haitalliset yhteydet. Esimerkiksi tietokantaan voi antaa pääsyn vain sitä käyttävälle podille ja muille tarvitseville palveluille. Toimeksiantaja tulee hyödyntämään NetworkPolicyä rajaamalla pääsyjä projektien podeissa.

9.4 RBAC

RBAC (Role-Based Access Control) antaa oikeuksia käyttäjille ja Service Accounteille roolien avulla. Tämän käyttöönoton jälkeen oikeuksia lisätään kahdella objektilla: Role ja RoleBinding. Role sisältää objektit ja verbsit/komennot (create, get, list, delete, jne), johon käyttäjällä on oikeudet tietyssä namespacessa. ClusterRolella annetaan oikeudet klusterin laajuisesti eli jokaiseen namespaceen. RoleBinding antaa oikeudet liittämällä Rolen käyttäjään tai Service Accountiin. RoleBindingin subject-kenttään laitetaan käyttäjä, ryhmä tai Service Account. RoleRef-kenttään puolestaan laitetaan liitettävä Role oikeuksineen. ClusterRoleBindingilla voi antaa oikeuksia klusterin laajuisesti.

RBAC otetaan käyttöön poistamalla käytöstä "Legacy Authorization" ja kun Kuberneteksen versio on 1.7.6. Legacy Authorization otetaan pois käytöstä seuraavalla komennolla, jossa kerrotaan klusterin nimi, sen sijainti ja optio poistosta:

gcloud container clusters update cluster-1 --zone europe-west3-a --no-enable-legacy-authorization

Kubernetes versioissa 1.6 ja 1.7 on bugi, jolloin Role- ja RoleBinding-objektien luontia ei voi tehdä ennen kuin itselleen on antanut cluster-adminin oikeudet. Tämä tehdään seuraavalla komennolla, jossa cluster-admin-test on ClusterRoleBindingin nimi ja user klusterin käyttäjä:

kubectl create clusterrolebinding cluster-admin-test --clusterrole=cluster-admin --user=<käyttäjän_sähköposti>

Cluster-adminiin liittäminen antaa käyttäjälle oikeudet koko Kubernetes klusteriin. Esimerkiksi hakea objektien tietoja, listata, luoda ja poistaa objekteja. Jokaisella namespacella on oletuksena Service Account nimellä default. RBACin ollessa käytössä annetaan oikeudet testispace1:n namespacelle testaen toimivuutta. Kuviossa 25 on ClusterRoleBinding-objekti, jossa roleRefin alla annetaan cluster-adminin oikeudet ja subjectsin alla kohdistetaan ne testispace1 namespacen default Service Accountille.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: default-sa-testispace1
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: default
  namespace: testispace1
```

Kuvio 25. ClusterRoleBinding antaa oikeuksia default Service Accountille

Testataan testispace1 namespacen default Service Accountin oikeuksia. Nginx-podi sijaitsee testispace1:ssä ja se listaa default namespacen podit alla olevalla komennolla. Se käyttää curl toimintoa ja autentikoi käyttäen default Service Accountin tokenia, joka tulee oletuksena jokaiselle podille, jos sitä ei olla ylikirjoitettu podin asetuksista. Osoitteena käytetään Kuberneteksen API-komponenttia, josta voi hakea tietyt objektit:

curl -ik -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" https://kubernetes.default.svc.cluster.local/api/v1/namespaces/default/pods

Tuloksena tulee lista default namespacen podeista eli oikeuksien antaminen onnistui. Jos testispace2 namespacessa jokin podi yrittää tehdä saman komennon, tulos kertoo, että se ei voi listata podeja, koska namespace testispace2:n Service Account default on tuntematon:

User "system:serviceaccount:testispace2:default" cannot list pods in the namespace "default".: "Unknown user \"system:serviceaccount:testispace2:default\""

Service Accountin asetuksiin on mahdollista lisätä *"automountServiceAccountToken: false"*, mikä ei oletuksena mountaa Service Accountin tokenia podeille. Ratkaisuna on luoda oma Service Account ja luoda RBACilla oikeudet tarvitseviin toimintoihin ja objekteihin.

Toimeksiantaja hyötyy RBACin tutkimisesta ja testaamisesta, koska Kubernetes tulee laajentumaan, jonka seurauksena oikeuksia joudutaan jakamaan vahinkojen ja tietoturvauhkien välttämiseksi. RBAC on Kuberneteksen paras vaihtoehto, kun määritetään tarvitut pääsyt käyttäjille ja Service Accounteille.

10 Kubernetes On-premise

10.1 Yleistä

Kubernetes On-premisesta on tietoturvasta asiat erikseen, koska se eroaa paljon Googlen tarjoamasta Container Enginestä. Google pitää itse huolta klusterien tietoturvasta, joiden muokkaamiseen ei ole käyttäjällä oikeuksia. Kuberneteksen asennus on-premisenä tarjoaa muokattavuutta enemmän, mutta myös hallinnan tarve kasvaa.

Admission Controllers ovat pluginin kaltaisia asetuksia, jotka ottavat vastaan tiettyjä kutsuja klusteriin. Nämä asetetaan master nodella päälle tai pois ja osa näistä ei ole päällä GKE:llä vaan Google päättää mitkä ovat käytössä. Esimerkiksi aikaisemmin mainitut ResourceQuota ja ServiceAccount laitetaan Admission Controllerista päälle. Toinen esimerkki on PodSecurityPolicy, jota ei voi ottaa käyttöön GKE:llä. Sillä olisi mahdollista asettaa pakotetut arvot objekteja luodessa esimerkiksi ei voi ajaa konttia root-käyttäjänä. GKE:stä ei löytynyt tietoa, mitkä Admission Controllerit ovat käytössä.

Jos toimeksiantaja ottaa Kubernetes On-premisen käyttöön tulevaisuudessa, se voi hyödyntää sille tarkastettuja tietoturvakäytänteitä.

10.2 CIS Kubernetes

CIS (Center for Internet Security) on tehnyt benchmarkeja eri tuotteista, joissa käydään läpi suositeltuja asetuksia. Benchmarkit ovat ladattavissa pdf-tiedostoina heidän verkkosivuillaan. Kuberneteksestä on tehty kaksi kappaletta, joista löytyy hyvin käytännöllisiä kohtia. Benchmarkissa käydään läpi muun muassa master ja worker noden konfiguraatiota liittyen niiden sisältämiin komponentteihin. Esimerkiksi onko kubelet nimisellä tiedostolla omistusoikeudet root-käyttäjällä. Benchmark sisälsi paljon oleellisia suosituksiakin kuten namespacen käyttöä. Suositukset käytiin läpi, mutta ei testattu puuttuvan On-premise palvelimen syystä. Benchmarkin voi myös suorittaa työkalulla, joka tarkistaa ovatko sen suositukset käytössä.

CIS Kubernetes Benchmark on hyödyllistä ottaa huomioon jatkossa, jos Kubernetes On-premise ratkaisua tullaan käyttämään toimeksiantajalla. Todennäköistä on, että benchmarkeja tulee jatkossa lisää, kun Kubernetestä kehitetään eteenpäin. Työkalua tarkistukseen voidaan hyödyntää myös jatkossa, koska ne ovat ladattavissa suoraan Githubista.

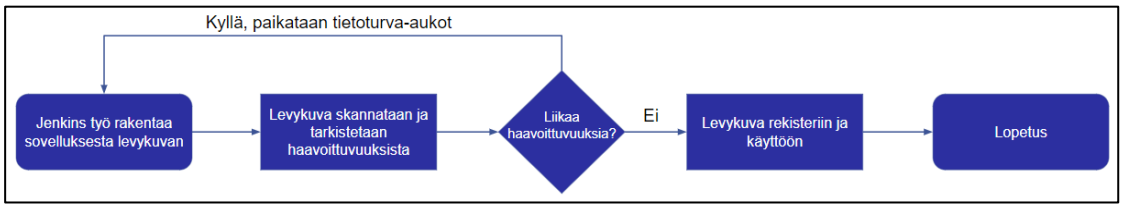
11 Haavoittuvuusskannerin integrointi

11.1 Lähtökohdat

Toimeksiantajalla on GKE:llä klusteri A, joka sisältää Jenkins podin ja sen luomat Jenkins slavet. Nämä tekevät Jenkinsin buildit projektien koodeista, jotka sisältävät ohjelman rakennuksen ja testauksen tietyllä ohjelmointikielellä, ohjelman rakennuksen Docker imageen, sen työntämisen Google Container Registryyn ja lopuksi projektin deploamisen Kuberneteksen klusteriin B. Docker imagen rakennuksen jälkeen pitäisi suorittaa haavoittuvuusskannaus ennen kuin image työnnetään rekisteriin. Jenkins hyödyntää Kubernetes pluginia, joka luo Jenkins slaven per buildityö ja suorittaa komennot Jenkinsfilestä. Siinä hyödynnetään Jenkins Shared Librarya, joka sisältää groovy skriptejä jokaiselle pipelinelle. Projektien koodit ja Jenkins Shared Library sijaitsevat toimeksiantajan versionhallinnassa.

Clair ja Postgres kontit laitetaan samaan podiin klusteriin A, että niihin saadaan yhteys Jenkins slavelta. Jenkins buildissa Docker imagen rakennuksen ja työntämisen rekisteriin hoitaa docker-client, joka hyödyntää Docker daemonin sockettia Kubernetes nodesta. Toimeksiantajan kanssa sovittiin Clairctl binääritiedoston rakennusta omassa Jenkins buildissa, jossa luotu binääri liitetään osaksi docker-clientia. Clairctl:n helppokäyttöisyyden vuoksi luotiin myös skripti tekemään haavoittuvuusskannaus, joka toimii jokaisella Jenkins buildilla.

Kuviossa 26 on Jenkins työn prosessikaavio, jossa käytetään Clairctl:n haavoittuvuusskannausta.



Kuvio 26. Jenkins työn prosessikaavio

11.2 Clairin asennus Kubernetekseen

Clairin asennus klusteriin A hoidetaan muutamalla Kuberneteksen objektilla ja komentorivikomennolla. Ensiksi luodaan Clairin objekteille oma namespace komennolla:

kubectl create namespace clair

Ensimmäisenä objektina on Service, joka ohjaa Clairiin tulevan liikenteen. Kuviossa 27 on sen YAML-tiedosto, jonka nimi on *clair-svc*, namespace *clair*, labelina app:clair, portteina 6060 ja 6061 sekä selectorina *app:clair*.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: clairsvc
  namespace: clair
  labels:
    app: clair
spec:
  type: NodePort
  ports:
  - port: 6060
    protocol: TCP
    nodePort: 30060
    name: clair-port0
  - port: 6061
    protocol: TCP
    nodePort: 30061
    name: clair-port1
  selector:
    app: clair
```

Kuvio 27. Clair Service YAML-tiedosto

Tämä otetaan käyttöön komennolla, jossa kerrotaan YAML-tiedoston nimi:

kubectl create -f clair-svc.yaml

Toisena on Clairin konfiguraatiotiedosto, jossa on asetukset Postgres-tietokantaan yhdistämiseen ja Clairin muita asetuksia. Tiedosto ei ole Kubernetes-objekti, mutta se luodaan Secretiksi. Kuviossa 28 on Clairin konfiguraatiotiedosto *config.yaml*, jonka alussa kerrotaan käytettävä tietokanta ja asetukset. Siinä kerrotaan tietokannan osoite, joka on localhost, koska Clair ja Postgres ovat samassa podissa. Sen jälkeen kerrotaan Postgresin perusasiat kuten portti, käyttäjä ja salasana. Sslmode pidetään pois päältä, koska yhteyksien ei tarvitse olla salattuja. Statement_timeout kertoo, kuinka monta millisekuntia odotetaan tietokantaan tulevia komentoja. Updaterin interval 6 h kertoo, kuinka usein Postgres-tietokanta päivitetään CVE-tiedoilla, jotka Clair hakee. Notifikaatioita ja avaintiedostoja ei käytetä, joten ne jäävät tyhjiksi.

```
clair:
  database:
    type: pgsql
    options:
      source: host=localhost port=5432 user=postgres
              password=passwordhere sslmode=disable statement_timeout=60000
    cacheSize: 16384

  api:
    port: 6060
    healthport: 6061

    timeout: 900s

    paginationKey:

    servername:
    cafile:
    keyfile:
    certfile:

  updater:
    interval: 6h

  notifier:
    attempts: 3
    renotifyInterval: 2h

    http:
      endpoint:
      servername:
      cafile:
      keyfile:
      certfile:
```

Kuvio 28. Clairin konfiguraatiotiedosto

Config.yaml muutetaan Secretiksi komennolla, jossa Secretin nimi on *clairsecret*, ja from-filella kerrotaan *config.yamlin* sijainti:

kubectl create secret generic clairsecret --from-file=config.yaml -n clair

Kolmantena luodaan Secret, joka sisältää salasanan Clairille Postgres-tietokantaan pääsemiseksi. Tämä viitataan viimeisessä Deployment-objektissa myöhemmin. Komennossa Secretin nimi on *postgres-sala* ja from-literalissa tehdään *key=value*-tyylinen data:

kubectl create secret generic postgres-sala --from-literal=supas=passwordhere -n clair

Viimeisenä on Deployment-objekti, joka sisältää Clair- ja Postgres-konttien asetukset sekä viittauksen edellisessä komennossa luotuun Secretin salasanaan. Kuviossa 29 on Deployment-objekti nimellä *clair-kubernetes.yaml*, jossa nimenä on *clair-and-postgres* ja replicas on podien määrä eli yksi riittää. Labelina on *app:clair*, jonka aiemmin

luotu Service *clairsvc* valitsee liikenteen kulkuun. Volumessa kerrotaan Clairin konfiguraatiotiedosto viittaamalla Secretin *clairsecret* ja nimenä *secret-volume*. Clair kontin alla määritetään Docker imageksi Clairin viimeisin versio *quay.io/coreos/clair:latest* ja käynnistysparametreissa käyttämään konfiguraatiotiedostoa polusta */config/config.yaml*. Tämä tiedosto liitetään *volumeMountsilla* viitateen aiemmin mainittuun *secret-volumeen*. Clairin kontin portit asetetaan *containerPortilla* eli 6060 ja 6061. Postgresin Docker imagena käytetään *postgres:9.6* ja ympäristömuuttuja *POSTGRES_PASSWORD* on aiemmin mainittu Secret *postgres-salan* salasana. Tämä ympäristömuuttuja on salasana, jolla Postgresiin voi yhdistää. Oletuksena käyttäjä on postgres, jote sitä ei tarvitse asettaa. Postgres kontin portti laitetaan myös *containerPortilla* eli 5432.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: clair-and-postgres
  namespace: clair
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: clair
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: clairsecret
      containers:
      - name: clair
        image: quay.io/coreos/clair:latest
        args:
          - "-config"
          - "/config/config.yaml"
        ports:
        - containerPort: 6060
        - containerPort: 6061
        volumeMounts:
        - mountPath: /config
          name: secret-volume
      - name: postgres
        image: postgres:9.6
        env:
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-sala
              key: supas
        ports:
        - containerPort: 5432
          name: postgres-port
```

Kuvio 29. Clair Deployment YAML-tiedosto

Tämä Deployment-objekti luotiin komennolla:

kubectl create -f clair-kubernetes.yaml

Clairilla menee 20-30 min, että tietokantaan on ladattu tunnetut CVE-tiedot.

11.3 Clairctl Jenkins-prosessiin

Clairctl binäärin tuominen docker-clientiin tehdään omalla Jenkins buildilla. Aluksi käytettiin alkuperäistä docker-clientiä, joka on julkisesti käytettävissä Docker Hubissa nimellä *ptcos/docker-client*. Sen tarkoitus on rakentaa Docker image, joka sisältää samat asiat ja Clairctl:n. Docker-clientin Dockerfile on esitetty kuviossa 30. Pohjaimagena käytetään *alpine.3.5*, joka on todella kevyt Linux-käyttöjärjestelmä. Ympäristömuuttuja *DOCKER_HOST:lla* asetetaan Docker daemonin sijainti. Argumenteissa on asetettu Docker clientin asennusta varten Dockerin asennuskansio, versio, latausosoite ja ryhmä ID sekä kubectl:n asennusta varten sen versio. Root-käyttäjänä päivitetään käyttöjärjestelmän paketit, asennetaan openssh-client, Docker client ja kubectl. Dockerille ja kubectl:lle annetaan suoritusoikeudet chmodilla. Lopussa lisätään docker ryhmä käyttäen alussa olevaa *DOCKER_GID:iä* ja liitetään uuteen jenkins käyttäjään, mikä antaa oikeuden käyttää Kuberneteksen Docker daemonia. Jenkins-käyttäjä luotiin käyttäjä ID:llä 10000.

```
FROM alpine:3.5

ENV DOCKER_HOST unix:///var/run/docker.sock
ARG DOCKER_PATH="/usr/bin"
ARG DOCKER_VERSION="1.11.2"
ARG DOCKER_URI="https://get.docker.com/builds/Linux/x86_64/docker-${DOCKER_VERSION}.tgz"
ARG DOCKER_GID="412"
ARG KUBECTL_VERSION="1.7.2"

USER root

RUN apk update && \
    apk add curl openssh-client && \
    rm -rf /var/cache/apk/* && \
    curl ${DOCKER_URI} -o /tmp/docker-${DOCKER_VERSION}.tgz && \
    cd /tmp && \
    curl ${DOCKER_URI}.sha256 -o - | sha256sum -c - && \
    tar -xvzf /tmp/docker-${DOCKER_VERSION}.tgz docker/docker && \
    mv -v docker/docker ${DOCKER_PATH}/docker && \
    rmdir -v docker && \
    rm -v /tmp/docker-${DOCKER_VERSION}.tgz && \
    chmod -v +x ${DOCKER_PATH}/docker && \
    curl -L https://storage.googleapis.com/kubernetes-release/release/v${KUBECTL_VERSION}/bin/linux/amd64/kubectl -o /usr/local/bin/kubectl && \
    chmod -v +x /usr/local/bin/kubectl && \
    addgroup -S -g ${DOCKER_GID} docker && \
    adduser -S -G docker docker && \
    adduser -G docker -u 10000 -D jenkins && \

USER jenkins
```

Kuvio 30. Alkuperäisen docker-clientin Dockerfile

Toimeksiantajan Githubiin luotiin uusi julkinen repository, jossa on uuden docker-clientin luontiin tarvittavat tiedostot kuten Dockerfile ja Jenkinsfile. Github repository löytyy osoitteesta https://github.com/protacon/ci-image-vulnerability-scan ja sen tiedostot ovat:

- Dockerfile, uuden rakennettavan Docker-clientin rakennusohje
- Jenkinsfile, Jenkins buildin ohje vaiheilla
- clairctl.groovy, haavoittuvuusskannauksen skripti
- clairctl.yml, Clairctl:n konfiguraatiotiedosto ajamiseen

Uusi Dockerfile on kuviossa 31, johon on lisätty Clairctl:n toimintaa varten ympäristömuuttuja *DOCKER_API_VERSION 1.23*, koska Clairctl toimi vain tuolla tietyllä Docker versiolla Kuberneteksessä. Ennen käyttöjärjestelmän päivitystä lisätään clairctl binääri, konfiguraatiotiedosto ja ajettava skripti. Lopuksi annetaan suoritusoikeudet clairctl binäärille ja ajettavalle skriptille.

```
FROM alpine:3.5

ENV DOCKER_HOST unix:///var/run/docker.sock
ARG DOCKER_PATH="/usr/bin"
ARG DOCKER_VERSION="1.11.2"
ARG DOCKER_URI="https://get.docker.com/builds/Linux/x86_64/docker-${DOCKER_VERSION}.tgz"
ARG DOCKER_GID="412"
ARG KUBECTL_VERSION="1.7.2"
ENV DOCKER_API_VERSION="1.23"

USER root

ADD $WORKSPACE/clairctl /usr/local/bin/
ADD clairctl.yml clairctl.groovy /usr/share/

RUN apk update && \
    apk add curl openssh-client && \
    rm -rf /var/cache/apk/* && \
    curl ${DOCKER_URI} -o /tmp/docker-${DOCKER_VERSION}.tgz && \
    cd /tmp && \
    curl ${DOCKER_URI}.sha256 -o - | sha256sum -c - && \
    tar -xvzf /tmp/docker-${DOCKER_VERSION}.tgz docker/docker && \
    mv -v docker/docker ${DOCKER_PATH}/docker && \
    rmdir -v docker && \
    rm -v /tmp/docker-${DOCKER_VERSION}.tgz && \
    chmod -v +x ${DOCKER_PATH}/docker && \
    curl -L https://storage.googleapis.com/kubernetes-release/release/v${KUBECTL_VERSION}/bin/linux/amd64/kubectl -o /usr/local/bin/kubectl && \
    chmod -v +x /usr/local/bin/kubectl && \
    addgroup -S -g ${DOCKER_GID} docker && \
    adduser -S -G docker docker && \
    adduser -G docker -u 10000 -D jenkins && \
    chmod -v +x /usr/local/bin/clairctl && \
    chmod -v +x /usr/share/clairctl.groovy

USER jenkins
```

Kuvio 31. Uuden docker-clientin Dockerfile

Kuviossa 32 on Clairctl:n konfiguraatiotiedosto clairctl.yml, jossa on asetettu käytettävät Clairin portit, Clairin sijainti ja HTML-raporttien paikka. Koska Kuberneteksessä podien IP-osoitteet vaihtelevat, niitä ei voida käyttää. DNS-nimenä käytetään Kuberneteksen metodia, jossa *clairsvc* on Servicen nimi, *clair* on namespacen nimi ja *svc.cluster.local* viittaa nykyiseen klusteriin.

```
clair:
  port: 6060
  healthPort: 6061
  uri: http://clairsvc.clair.svc.cluster.local
  report:
    path: ./reports
    format: html
```

Kuvio 32. clairctl.yml konfiguraatiotiedosto

Kuvion 33 Jenkinsfilella asetetaan mitä Jenkins tekee työssään. PodTemplatessa asetetaan Jenkins slavepodin käytettävät kontit Containersin alla, missä jokaisella kontilla on containerTemplate. Alleviivattu *label:'clairctl'* kerrotaan Jenkins pipelinen nodessa, kun käytetään kyseistä podia. Ensimmäinen kontti on nimeltään golang ja Docker imagella *golang:1.8*, jota käytetään Clairctl binäärin luonnissa. *TtyEnabled:true* sallii tulostusten näkymisen Jenkins työn konsolilla ja *command: '/bin/sh -c', args: 'cat'* pitää kontin käynnistyksen jälkeen päällä. Toisena konttia käytetään vanhaa docker-clientia *ptcos/docker-client:latest*, joka aina haetaan uudestaan asetuksella *alwaysPullImage:true*. Volumes-kohdassa mountataan kontteihin Docker socketti Kuberneteksen nodesta. *Node('clairctl')* käyttää podTemplaten asetuksia ja ajaa lopun Jenkinsfilen podissa. Staget kertoo Jenkins pipelinen eri vaiheet ja stagessa checkout komento *"checkout scm"* hakee annetun työn koodin versionhallinnasta, kertoo käytettävän commitin ja sen viestin.

Build stagessa käytetään golang-konttia, joka ajaa Clairctl binäärin asennukseen tarvittavat komennot. Ensiksi go-wrapper hakee Clairctl:n koodin Githubista, haetaan tarvittava riippuvuus go-bindata ja luodaan Clairctl binääri komennolla:

CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o /go/bin/clairctl github.com/jgsqware/clairctl

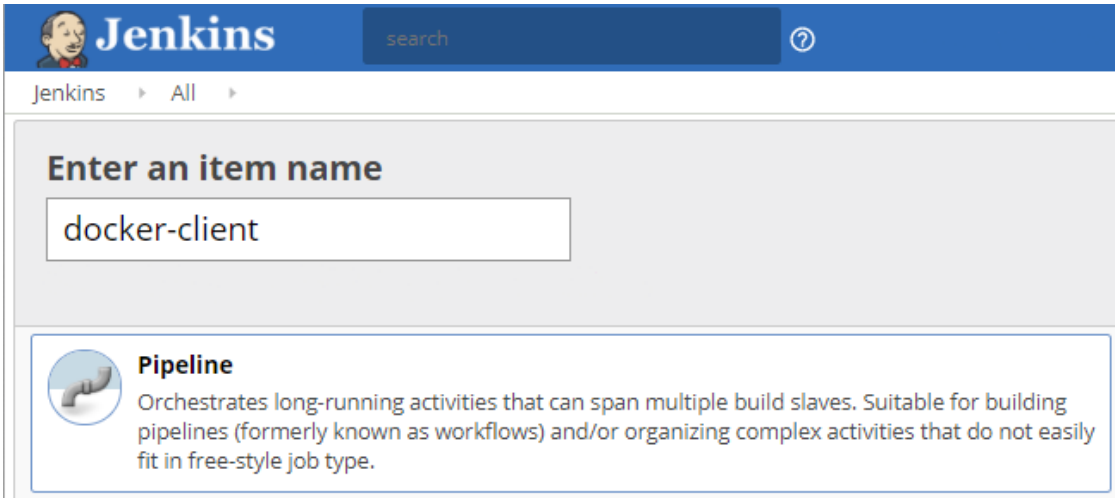
Komennon osa *CGO_ENABLED=0* luo binäärin ilman dynaamista cgo-binääriä vaan käyttäen staattista binääriä, koska muuten clairctl binäärin ajo ei toimi. GOOS=linux luo binäärin Linuxille käytettäväksi. Paketit rakennetaan uudestaan ilman cgo:ta komennon osalla *"-a -installsuffix cgo"* ja polulle */go/bin/clairctl* luodaan binääri. Lopuksi golang-kontilla siirretään binääri Jenkinsin workspacelle, jota käytetään Jenkins työn lopussa.

Package stagessa käytetään docker-clientia ja asetetaan käyttämään Jenkinsin asetuksista Docker Hubin repositoryn tunnuksia. Docker rakentaa uuden docker-clientin Docker imagen nimillä *ptcos/docker-client:latest* ja *ptcos/docker-client:1.1.$BUILD_NUMBER*. Ensimmäinen viittaa uusimpaan docker-clientiin ja toinen tagitaan versiolla 1.1.X, jossa X on Jenkins työn nro. Lopuksi molemmat työnnetään Docker Hubin rekisteriin.

```
podTemplate(label: 'clairctl',
    containers: [
        containerTemplate(name: 'golang', image: 'golang:1.8', ttyEnabled: true, command: '/bin/sh -c', args: 'cat'),
        containerTemplate(name: 'docker', image: 'ptcos/docker-client:latest', alwaysPullImage: true, ttyEnabled: true, command: '/bin/sh -c', args: 'cat')
    ],
    volumes: [
        hostPathVolume(hostPath: '/var/run/docker.sock', mountPath: '/var/run/docker.sock')
    ]
) {
    node('clairctl') {
        stage('Checkout') {
            checkout scm
        }
        stage('Build') {
            container('golang') {
                sh """
                go-wrapper download github.com/jgsqware/clairctl
                cd /go/src/github.com/jgsqware/clairctl
                go get -u github.com/jteeuwen/go-bindata/...
                go generate ./clair
                cd /go
                CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o /go/bin/clairctl github.com/jgsqware/clairctl
                mv -f /go/bin/clairctl $WORKSPACE/clairctl
                """
            }
        }
        stage('Package') {
            container('docker') {
                docker.withRegistry('https://registry.hub.docker.com','docker-hub-credentials') {
                    sh """
                    docker build -t ptcos/docker-client:latest -t ptcos/docker-client:1.1.$BUILD_NUMBER .
                    """
                    def image = docker.image("ptcos/docker-client")
                    image.push("latest")
                    image.push("1.1.${env.BUILD_NUMBER}")
                }
            }
        }
    }
}
```

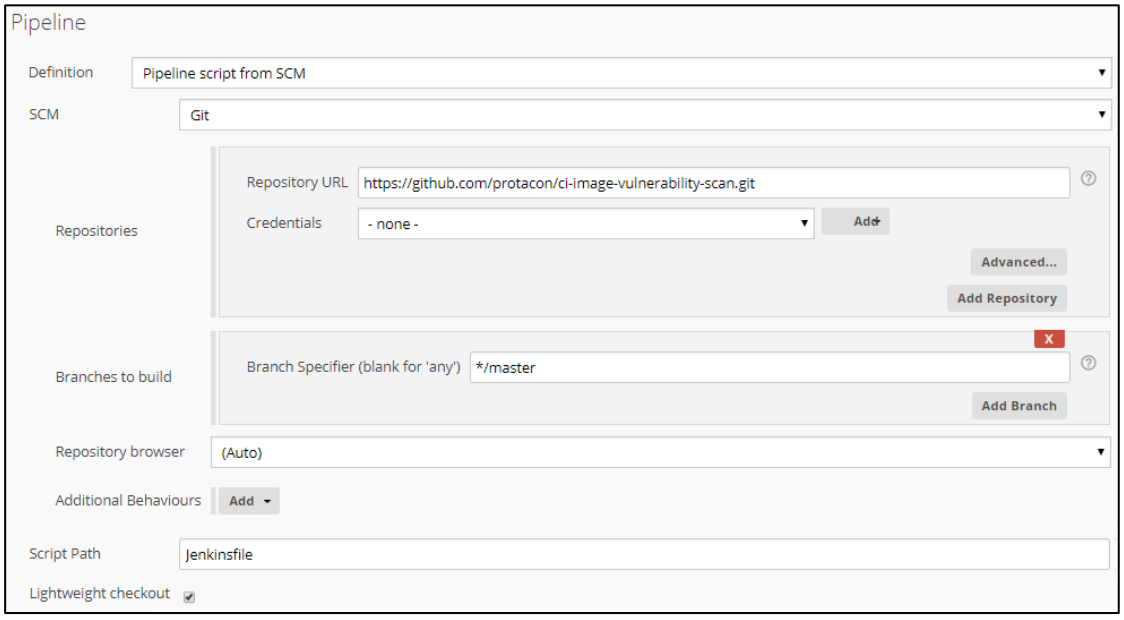
Kuvio 33. Jenkinsfile uuden docker-clientin luontiin

Luodaan docker-clientille Jenkins työ toimeksiantajan Jenkins-palvelimelle. Docker-clientin Jenkins työ luodaan myös siksi, että tiedostojen muuttuessa sen voi muutkin rakentaa uudestaan. Kuviossa 34 luodaan uusi Jenkins työ Pipeline-projektina ja nimetään docker-clientiksi.



Kuvio 34. Jenkins työn luonti docker-clientista

Kuviossa 35 muokataan luodun Jenkins työn asetukset. Pipeline definitionissa määritetään Pipeline skriptin sijainti, joka löytyy versionhallinnasta. SCM:ksi valitaan *Git* ja repositoryyn *protacon/ci-image-vulnerability-scan*. Script Path on Jenkinsfile, koska se ei ole kansion sisällä vaan suoraan repositoryn polun alussa.



Kuvio 35. Docker-client Jenkins työn asetukset

Tämän Jenkins työn ajon jälkeen Docker Hubissa on uusi docker-client, joka sisältää Clairctl binäärin ja sen ajoon tarvitsemat tiedostot.

HTML-raporttien säilyttämiseen asennetaan HTML Publisher Plugin, joka ottaa talteen HTML-tiedostot halutusta Jenkins työn kansiosta ja pitää niitä muistissa. Ilman tätä Clairctl:n luomat raportit jäävät Jenkins slaven konttiin ja katoavat kontin poistossa. Jenkins ei oletuksena salli CSS-tyyliä, jota käytetään Clairctl:n raporteissa. Jenkinsin käynnistysoptioihin asetettiin ympäristömuuttuja *JAVA_OPTS:iin* seuraava rivi, jolloin sallitaan CSS-tyylit alkuperäisestä asetuksesta:

-Dhudson.model.DirectoryBrowserSupport.CSP="style-src 'self' 'unsafe-inline';"

Tämän testaus onnistuu myös kirjoittamalla Jenkinsin Script Consoleen seuraava komento, mutta se ei uudelleen käynnistyksen jälkeen ole enää käytössä:

System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "style-src 'self' 'unsafe-inline';")

Jenkinsin työn kuvaukseen lisätään haavoittuvuusskannauksen tulokset HTML-muodossa. Tämä pitää sallia Jenkinsin asetuksista *Manage Jenkins -> Configure Global Security* ja sieltä Markup Formatteriin valitaan *Safe HTML*, kun aikaisemmin oli *Plain text*.

11.4 Testityö

Uusi docker-client testataan Jenkinsissä erillisellä testaustyöllä ilman uuden Docker imagen rakennusta ja työntöä Google Container Registryyn eli oikeat työt poikkeavat vähän. Tätä käytetään myös jatkossa erilaisiin Jenkins testeihin.

Sitä ennen käydään haavoittuvuusskannauksen *clairctl.groovy* skripti liitteestä 1. Skriptin alussa tuodaan rakennetun Docker imagen nimi skannattavaksi ja käytetään sitä määrittämällä *PTCS_DOCKER_REGISTRY_ANALYSIS*, mutta /-merkit korvataan viivoilla. Esimerkiksi *repository/docker-image* on *repository-docker-image*. Tämä

tehdään HTML-raportin tarkistusta varten. Määritetty *FIXEDJOBNAME* käytetään HTML-raportin linkissä. Siinä korvataan /-merkki sanalla */job/*. Jenkins pipeline työn nimi on *työnimi/master*, jolloin HTML-raportin linkki ei toimi Jenkinsissä. Kun /-merkki korvataan ja työn nimi on skriptissä *työnimi/job/master*, linkki toimii. Normaaleissa Jenkins töissä linkki on *jenkinsin-osoite/job/työnimi/raportin-nimi/raportti.html* ja pipeline töissä *jenkinsin-osoite/job/työnimi/job/master/raportin-nimi/raportti.html*.

Clairctl lähettää lokaalin Docker imagen layerit Clairiin käyttäen konfiguraatiotiedostoa *clairctl.yml*, minkä jälkeen se tarkastaa Docker imagen sisältävät haavoittuvuudet ja luo HTML-raportin. Jenkins ottaa HTML-raportin talteen HTML Publisher Pluginilla. Eri haavoittuvuudet lasketaan HTML-raportista vakavuuden mukaan käyttäen grep:iä. Raportti esittelee jokaisen haavoittuvuuden muodossa * haavoittuvuusluokka *, jolloin haavoittuvuudet saadaan selville. Tulokset trimmataan ja echolla kerrotaan tulokset jokaisesta haavoittuvuusluokasta. Tulokset tallennetaan ympäristömuuttujiin ja määritetään integereiksi tulevia haavoittuvuuksien määrien tarkastusta varten. Myös haavoittuvuuksien rajat esitellään Jenkins ympäristömuuttujista, jotka on asetettu Jenkinsin asetuksista. Seuraavat if/else blockit tarkastavat haavoittuvuuksien luokkien mukaan, onko haavoittuvuuksia enemmän kuin määritetty raja. Jos haavoittuvuuksia on yli rajan, Jenkins työ keskeytyy. Jos haavoittuvuuksia on alle rajan, Jenkins työ jatkuu. Lopuksi tulokset tallennetaan muistiin ympäristömuuttujiin tulevaa report groovy skriptiä varten, että ne näkyvät Jenkins työn kuvaksessa.

Kuviossa 36 on testityön Dockerfile versionhallinnassa, joka käyttää Jenkins Shared Librarya nimeltä *PTCSLibrary@1.0.0* ja docker-clientin versiota 1.1.32. *Def project = 'docker-client'* kertoo skannattavan Docker imagen nimen, mitä käytetään *clairctl.groovy* skriptissä. Koodi tarkistetaan ja käytetään *testContainer* skriptiä, johon tuodaan def:illa mainittu *project* Jenkins Shared Librarystä. Sieltä myös käytetään *report.FlushReport* groovy skriptiä, joka laittaa Jenkins työn kuvaukseen haavoittuvuuksien määrät.

```
@Library("PTCSLibrary@1.0.0") _

podTemplate(label: 'clairctl',
    containers: [
        containerTemplate(name: 'docker', image: 'ptcos/docker-client:1.1.32', alwaysPullImage: true, ttyEnabled: true, command: '/bin/sh -c', args: 'cat')
    ]
) {
    def project = 'docker-client'

    node('clairctl') {
        stage('Checkout') {
            checkout_with_tags()
        }
        stage('Testing') {
            container('docker') {
                def published = testContainer(project);
            }
        }
    }
}

report.FlushReport()
```

Kuvio 36. Jenkins testityön Jenkinsfile

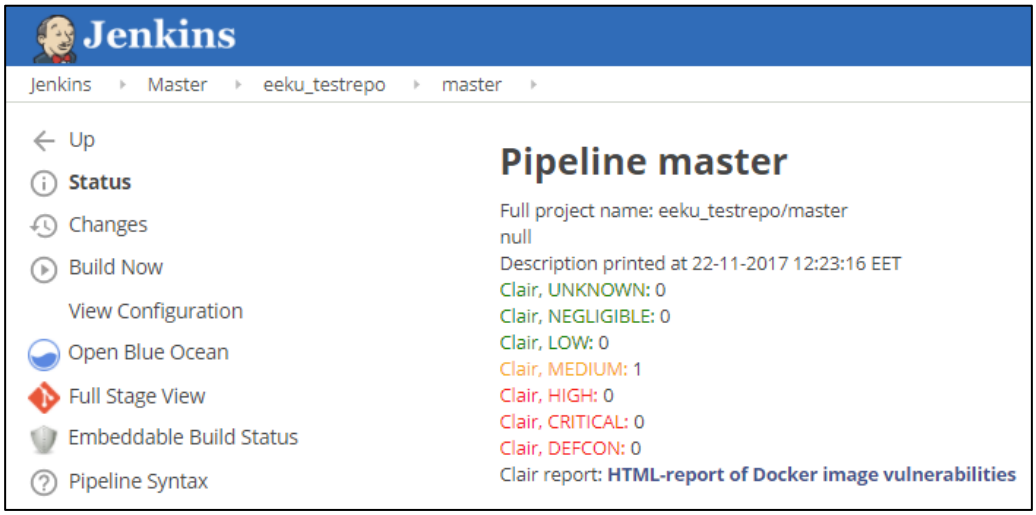
Kuviossa 37 on *testContainer* groovy skripti, joka ajetaan testityössä. *Def fullPath = "ptcos/$projectName"* käytetään Docker imagen lopullisena nimenä clairctl groovy skriptissä. Docker-clientin polusta */usr/share/clairctl.groovy* kopioidaan skripti Jenkinsin workspaceen, koska Jenkins ei osannut ajaa skriptiä muualta. Skripti ladataan ja sinne viedään *fullPath*. *Report.WriteLinet* kirjoittavat ajettavan buildin kellonajan, haavoittuvuuksien määrät ja Clairin raportin paikan muistiin.

```
#!/usr/bin/groovy

def call(String projectName) {
        def fullPath = "ptcos/$projectName"
        sh "cp /usr/share/clairctl.groovy ${WORKSPACE}"
        clairRun = load 'clairctl.groovy'
        clairRun.vulnerabilitycheck(fullPath)
        report.WriteLine("""Description printed at ${BUILD_TIMESTAMP}""")
        report.WriteLine("""<font color="green">Clair, UNKNOWN: </font>""" + env.UNKNOWN_V)
        report.WriteLine("""<font color="green">Clair, NEGLIGIBLE: </font>""" + env.NEGLIGIBLE_V)
        report.WriteLine("""<font color="green">Clair, LOW: </font>""" + env.LOW_V)
        report.WriteLine("""<font color="orange">Clair, MEDIUM: </font>""" + env.MEDIUM_V)
        report.WriteLine("""<font color="red"> Clair, HIGH: </font>""" + env.HIGH_V)
        report.WriteLine("""<font color="red"> Clair, CRITICAL: </font>""" + env.CRITICAL_V)
        report.WriteLine("""<font color="red"> Clair, DEFCON: </font>""" + env.DEFCON_V)
        report.WriteLine("""Clair report: <a href=/job/${env.FIXEDJOBNAME}/Vulnerability_Report/analysis-${env.PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html><b>HTML-report of Docker image vulnerabilities</b></a>""")
```

Kuvio 37. TestContainer groovy skripti Jenkins Shared Librarysta

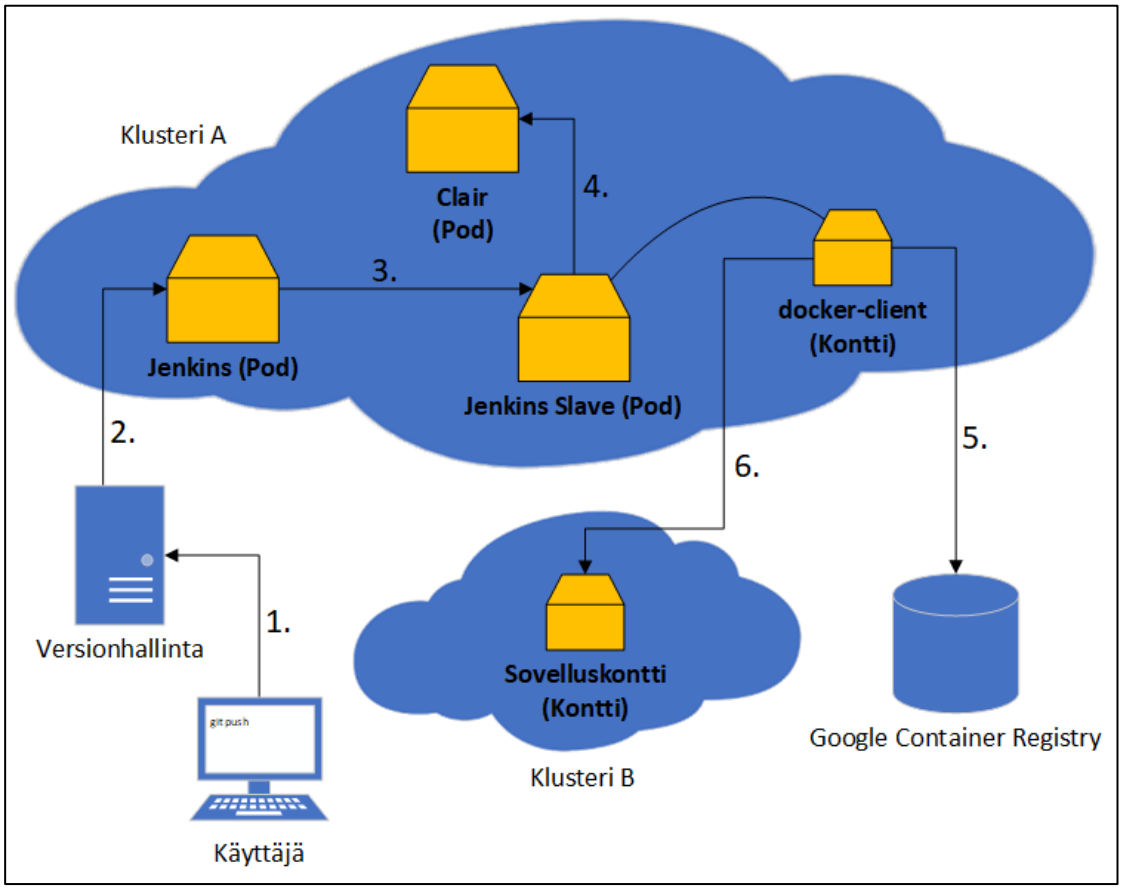
Kun *testContainer* groovy skripti on ajettu, testityön report.FlushReport() skripti ajetaan, mikä kirjoittaa Jenkins työn kuvaukseen muistissa olleet tiedot. Kuviossa 38 on Jenkins testityön kuvaus, jossa tekstin *Pipeline master* alla on tulostetut tiedot.



Kuvio 38. Jenkins työn kuvaus haavoittuvuuksineen ja linkki raporttiin

Kuviossa 39 on kokonaiskuva prosessista.

1. Käyttäjä työntää koodia ja Jenkinsfilen versionhallintaan
2. Versionhallinta huomaa Jenkinsfilen ja aloittaa sen pohjalta työn (projektin työ luodaan, jos sitä ei ole)
3. Jenkins luo työtä varten Jenkins slave podin
4. Slave podin docker-client tarkastaa Docker imagen lähettämällä sen layerit Clairiin
5. Docker image työnnetään Google Container Registryyn
6. Docker image otetaan käyttöön klusterissa B



Kuvio 39. Kokonaiskuva Jenkins prosessista

Toimeksiantaja hyötyy haavoittuvuuksien tietoisuudesta ja poistaa ne ennen tuotantoon sijoittamista tietoturvauhkien välttämiseksi.

12 Pohdinta

Opinnäytetyön tavoitteena oli selvittää keinoja tehdä tietoturvallinen Kubernetes konttiympäristö ja etsiä työkalu Docker imageiden haavoittuvuuksien skannaukseen. Tietoturvallisen Kuberneteksen keinot testattiin, jotka osoittautuivat tarpeellisiksi ja dokumentoitiin talteen muiden hyödyllisten tietojen lisäksi kuten Kubernetes On-premise ja CIS Kubernetes. Haavoittuvuusskannereita vertailtiin ja Clair+Clairctl asennettiin osaksi Jenkins CI/CD-prosessia. Haavoittuvuusskanneri tarkastaa vasta rakennetun Docker imagen ja luo HTML-raportin helppoa tarkastelua varten. Raportti

paljastaa haavoittuvuudet eri vakavuusluokilta, joka helpottaa paikkaamaan vakavimmat ensin. Tietoturvallisia keinoja etsittiin erityisesti Google Container Engineen, koska se oli jo käytössä. Ohjelmistotuotannon kanssa piti tehdä yhteistyötä, että heidän työnsä eivät vaikeudu tuloksien myötä esimerkiksi Jenkins Shared Library tuli kesken skriptien teon. Haavoittuvuusskannaukseen käytetyt osat Jenkinsissä ovat nähtävillä ja päivitetään osoitteessa https://github.com/protacon/ci-image-vulnerability-scan.

Haavoittuvuusskannereiden tutkiminen oli selkeä tavoite ja laadullinen vertailu oli käytetty tutkimusmetodologia. Se oli kaikista sopivin, koska jokaista skanneria testattiin alusta loppuun ilman aiempaa tietämystä niistä. Se auttoi valitsemaan oikean vaihtoehdon, mutta myös päättely Clairin tulevaisuudesta ja avun löytämisestä oli tärkeää lopullisen tuotteen valinnassa.

Melkein kaikki aiheiden lähdetieto haettiin verkosta. Yleisesti tietoturvasta löytyi tietoa kirjoista, mutta konttien tietoturvasta oli enemmän verkossa. Kuberneteksestä ei löytynyt ajankohtaisia kirjoja. Vaikka yhdestä keskeneräisestä Kuberneteksestä kertovasta e-kirjasta saatiin tietoa, osa siitäkin oli jo vanhentunutta. Aiempi Kubeneteksestä saatu kokemus toimeksiantajalta auttoi kertomaan vanhetuneen tiedon. Kuberneteksen verkkosivuilta löytyi enemmän tietoa ja ajankohtaisempia aiheita kuten päivittyneemmät ja paremmat Kubernetes-objektit. Eri blogikirjoitukset kertoivat uusista ominaisuuksista, jota julkistettiin keväällä. Varsinkin NetworkPolicy Calico kontrollerilla ja RBAC yleistyminen Google Container Enginellä. Githubin ja muiden avoimen lähdekoodin aiheiden tiedot olivat hyvin selitetty ja helposti ymmärrettävää, mitä käytettiin hyödyksi. Githubissa Clairctl:n tekijältä kysyttiin myös kerran neuvoa, kun se ei toiminut Docker versioiden yhteensopimattomuuden takia. Ongelma ratkeutui asettamalla tietty DOCKER_API_VERSION.

Työhön voi olla tyytyväinen, koska molemmat tavoitteet saavutettiin. Eniten vaikeuksia tuotti Clairctl:n integroiminen docker-clientiin. Eri versio-, kirjasto- ja käyttöjärjestelmäriippuvuudet hankaloittivat kokonaisuuden luontia. Varsinkin testausympäristön erot käytettävästä ympäristöstä. Skannereiden testauksen olisi voinut tehdä Kubernetes ympäristössä Centos 7-virtuaalikoneen sijasta, mikä tarkentaa tulosten realistisuutta. Kubernetes testit samassa ympäristössä erillisillä klustereilla oli hyvä esimerkki siitä. Niiden testaaminen meni myöhemmäksi, koska

Clairctl:n integroimiseen käytettiin enemmän aikaa. On-premise ratkaisuun liittyen olisi voitu vielä testata oman Kubernetes klusterin asennusta ja eri network plugineita mahdollista omaa ratkaisua varten.

Koska Kubernetekseen tulee jatkuvasti uusia päivityksiä, täytyy seurata niiden tuomia uusia ominaisuuksia. Nämä voivat parantaa tietoturvaa entistä enemmän. OWASP Dependency Checkerin Dagdaa voidaan hyödyntää jatkossa sovelluksien kirjastojen skannauksessa, kun Clair tarkastaa pakettien tiedot. Google Container Registryssä on Alpha-vaiheessa oma haavoittuvuusskannaus, jota voisi myös hyödyntää. Clairctl skriptin virheen tullessa se lopettaa Jenkins buildin esimerkiksi HTML-raportin teossa. Clairctl tehtiin jo virhesietoisemmaksi ja lähettää virheistä myös Slackiin yksityisviestin. Clairctl:n sallittujen haavoittuvuuksien rajoja täytyy vielä seurata, että ovatko ne hyvät vai keskeytetäänkö Jenkins työ vain kaikista kriittisimmistä haavoittuvuuksista kuten Critical ja Defcon1.

Opinnäytetyötä tehdessä Kubernetes ja Jenkins tulivat tutummaksi. Ongelmien ratkontaa oppi enemmän tarkastamalla lähdekoodista suoraan, kun vika oli paljon syvemmällä tai muualla kuin lokeissa. Docker oli aiemmin tullut tutuksi ja sen käyttämisessä ei ollut ongelmia. Kubernetes oli aivan uusi juttu tekijälle, mikä aluksi vaikeutti tietojen etsintää siihen liittyvästä tietoturvasta. Google Container Enginen ja Kubernetes On-premisen erot eivät olleet suoraan nähtävillä missään ja tuli ilmi vasta CIS Kubernetes Benchmarkkien tarkastuksessa, kun Kuberneteksen master nodeen ei voinut yhdistää ja worker nodeilla ei ollut tiettyjä suosituksia mahdollista tehdä.

Toimeksiantaja hyötyi käytännöllisistä keinoista tehdä tietoturvallisempi Kubernetes, mitä hyödynnetään jatkossa eri projekteissa. Kubernetes On-premise ratkaisua varten on dokumentoitu esitietoa toimeksiantajalle, joka auttaa oman Kubernetes ympäristön rakentamisessa. Haavoittuvuusskannerin avulla saadaan selville tunnetut tietoturvahaavoittuvuudet omista rakennetuista Docker imageista ja päivittämällä paketit poistetaan uhat tekemällä toimeksiantajan palveluista turvallisempia.

Lähteet

About CVE. 2017. Mitren websivut. Viitattu 3.7.2017. https://cve.mitre.org/about/

About images, containers, and storage drivers. 2017. Docker docs verkkosivuilla. Viitattu 1.8.2017. https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/

Bernstein, D. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing, 1, 3, 81-84. Viitattu 4.6.2017. http://ieeexplore.ieee.org/document/7036275/, IEEE.

Brown, N. 2016. Explaining Docker Image IDs. Windsock.io verkkosivuilla. Viitattu 2.8.2017. http://windsock.io/explaining-docker-image-ids/

Category:OWASP Top Ten Project. 2017. OWASPin websivut. Viitattu 6.7.2017. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Chesler, R. 2017. Types of Hypervisors (Type 1 and Type 2). Viitattu 23.7.2017. IT Connected verkkosivuilla. https://www.itconnected.tech/blog/types-of-hypervisorstype-1-and-type-2/

Clair. 2017. Clairin Github repository. Viitattu 4.6.2017. https://github.com/coreos/clair

Clairctl. 2017. Clairctl:n Github repository. Viitattu 15.6.2017. https://github.com/jgsqware/clairctl

Combe, T., Di Pietro, R. & Martin, A. 2016. To Docker or Not to Docker: A Security Perspective. IEEE Cloud Computing, 3, 5, 54-62. Viitattu 11.4.2017. http://ieeexplore.ieee.org/document/7742298/, IEEE.

Concepts. N.d. Kuberneteksen dokumentaation verkkosivut. Viitattu 5.10.2017. https://kubernetes.io/docs/concepts/

Confidentiality, Integrity, and Availability. 2016. Mozilla Developer Networkin verkkosivut. Viitattu 3.7.2017. https://developer.mozilla.org/en-US/docs/Web/Security/Information_Security_Basics/Confidentiality,_Integrity,_and_Availability

Container Engine. N.d. Google Cloud Platformin verkkosivut. Viitattu 12.10.2017. https://cloud.google.com/container-engine/

CVE ID Syntax Change. 2016. Mitren websivut. Viitattu 5.7.2017. https://cve.mitre.org/cve/identifiers/syntaxchange.html

Dagda. 2017. Dagdan Github repository. Viitattu 20.6.2017. https://github.com/eliasgranderubio/dagda

Daou, I. 2016. Type 1 vs. 2 Hypervisor Virtualization Platform. CCNA Hub verkkosivut. Viitattu 23.7.2017. https://www.ccnahub.com/linux/type-1-vs-2-hypervisor-virtualization-platform/

Deepfenceio/deepfence_depcheck. 2017. Deepfenceio:n Docker hub repository. Viitattu 20.6.2017. https://hub.docker.com/r/deepfenceio/deepfence_depcheck/

Desikan, T., Gummaraju, J. & Turner, Y. 2015. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities. Banyanops websivut. Viitattu 5.7.2017. https://www.banyanops.com/blog/analyzing-docker-hub/

Dockerfile reference. 2017. Docker docs verkkosivuilla. Viitattu 1.8.2017. https://docs.docker.com/engine/reference/builder/

Docker overview. 2017. Docker docs verkkosivuilla. Viitattu 9.8.2017. https://docs.docker.com/engine/docker-overview/

Ellingwood, J. 2017. An Introduction to Continuous Integration, Delivery, and Deployment. DigitalOceanin verkkosivuilla. Viitattu 24.8.2017. https://www.digitalocean.com/community/tutorials/an-introduction-to-continuous-integration-delivery-and-deployment

Garcia, J. 2016. Hyperclair usage. Video joka opastaa Clairctl työkalun käyttöä. Asciineman verkkosivuilla. Viitattu 18.6.2017. https://asciinema.org/a/41461

Hall, K. 2017. UK hospital meltdown after ransomware worm uses NSA vuln to raid IT. The Register verkkosivut. Viitattu 3.7.2017. https://www.theregister.co.uk/2017/05/12/nhs_hospital_shut_down_due_to_cyber_attack

Hirsjärvi, S., Remes, P. & Sajavaara, P. 2007. Tutki ja kirjoita. 13. p. Helsinki: Tammi

Historia. N.d. Protaconin verkkosivut. Viitattu 2.5.2017. https://www.protacon.com/historia/

ICT-palvelut. N.d. Protaconin verkkosivut. Viitattu 4.6.2017. https://www.protacon.com/digitalisaatio/ict-palvelut/

Integrations. 2017. Clairin Github repository. Viitattu 15.6.2017. https://github.com/coreos/clair/blob/master/Documentation/integrations.md

Jenkins. N.d. Jenkinsin verkkosivut. Viitattu 24.8.2017. https://jenkins.io/

Jenkins User Documentation. N.d. Jenkinsin dokumentaation verkkosivut. Viitattu 24.8.2017. https://jenkins.io/doc/

Kerttula, E. 2000. Tietoverkkojen tietoturva. 3. p. Helsinki: Oy Edita Ab.

Klar. 2017. Klarin Github repository. Viitattu 15.6.2017. https://github.com/optiopay/klar

Lukša, M. 2017. Kubernetes in Action MEAP Edition. Manning Publications.

Media. N.d Protaconin verkkosivut. Viitattu 2.5.2017. https://www.protacon.com/media/

Mehiläinen, T. 2017. Pilvipalvelut yrityksen tietoturvan kulmakivenä. Magic Cloud verkkosivut. Viitattu 3.7.2017. https://magiccloud.fi/pilvipalvelut-yrityksen-tietoturvan-kulmakivena/

Official Common Platform Enumeration (CPE) Dictionary. 2017. NIST:n NVD verkkosivut. Viitattu 20.6.2017. https://nvd.nist.gov/products/cpe

Overview of Docker Hub. 2017. Docker docs verkkosivut. Viitattu 8.8.2017. https://docs.docker.com/docker-hub/

Overview of kubectl. N.d. Kuberneteksen dokumentaation verkkosivut. Viitattu 10.10.2017. https://kubernetes.io/docs/user-guide/kubectl-overview/

OWASP Dependency Check. 2017. Owaspin verkkosivut. Viitattu 20.6.2017. https://www.owasp.org/index.php/OWASP_Dependency_Check

OWASP Top 10 Application Security Risks – 2017. 2017. OWASPin verkkosivuilla. Viitattu 6.7.2017. https://www.owasp.org/index.php/Top_10_2017-Top_10

Pipeline. N.d. Jenkinsin dokumentaation verkkosivut. Viitattu 5.9.2017. https://jenkins.io/doc/book/pipeline/

Pipeline as Code with Jenkins. N.d. Jenkinsin dokumentaation verkkosivut. Viitattu 5.9.2017. https://jenkins.io/solutions/pipeline/

Pod Overview. N.d. Kuberneteksen dokumentaation verkkosivut. Viitattu 10.10.2017. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

Protacon. N.d. Protaconin etusivut. Viitattu 2.5.2017. https://www.protacon.com/

Protacon Solutions Oy. N.d. Finderin verkkosivut. Viitattu 4.6.2017. https://www.finder.fi/IT-konsultointia+IT-palveluja/Protacon+Solutions+Oy/Jyv%C3%A4skyl%C3%A4/yhteystiedot/249088

Repositories on Docker Hub. 2017. Docker docs verkkosivut. Viitattu 8.8.2017. https://docs.docker.com/docker-hub/repos/

Secrets. N.d. Kuberneteksen dokumentaation verkkosivut. Viitattu 12.10.2017. https://kubernetes.io/docs/concepts/configuration/secret/

Services. N.d. Kuberneteksen dokumentaation verkkosivut. Viitattu 10.10.2017. https://kubernetes.io/docs/concepts/services-networking/service/

Sulinski, J. 2017. Docker Image Vulnerability Research. Blogipostaus Federacyn verkkosivuilla. Viitattu 5.7.2017. https://www.federacy.com/docker_image_vulnerabilities

Suomalaisten tietoja kalastellaan aktiivisesti. 2017. Uutinen Viestintäviraston verkkosivuilla 1.6.2017. Viitattu 3.7.2017. https://www.viestintavirasto.fi/kyberturvallisuus/tietoturvanyt/2017/06/ttn2017060111023.html

Takala, H. 2016. Tietoturvallisuus tähtäimessä. Netmanin verkkosivut. Viitattu 3.7.2017. https://www.netman.fi/fi/it-blogi/tietoturva/

Top 10 2017-A9-Using Components with Known Vulnerabilities. 2017. OWASPin websivut. Viitattu 6.7.2017. https://www.owasp.org/index.php/Top_10_2017-A9-Using_Components_with_Known_Vulnerabilities

Understanding Kubernetes Objects. N.d. Kuberneteksen dokumentaation verkkosivut. Viitattu 5.10.2017. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

Understanding virtualization. 2017. Redhatin verkkosivut. Viitattu 23.7.2017. https://www.redhat.com/en/topics/virtualization

USN-2097-1: curl vulnerability. 2014. Ubuntun verkkosivut. Viitattu 5.7.2017. https://www.ubuntu.com/usn/USN-2097-1/

Wallen, J. 2017. How to use Docker tags to add version control to images. Techrepublicin verkkosivut. Viitattu 8.8.2017. http://www.techrepublic.com/article/how-to-use-docker-tags-to-add-version-control-to-images/

Welcome to OWASP. 2017. OWASPin verkkosivut. Viitattu 6.7.2017. https://www.owasp.org/index.php/Main_Page

What is a Container. 2017. Dockerin verkkosivut. Viitattu 25.7.2017. https://www.docker.com/what-container

What is Docker. 2017. Dockerin verkkosivut. Viitattu 25.7.2017. https://www.docker.com/what-docker

What is Kubernetes? N.d. Kuberneteksen dokumentaation verkkosivut. Viitattu 5.10.2017. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

What is virtualization? 2017. Redhatin verkkosivut. Viitattu 23.7.2017. https://www.redhat.com/en/topics/virtualization/what-is-virtualization

Yhteystiedot. N.d. Protaconin verkkosivut. Viitattu 4.6.2017. https://www.protacon.com/yhteystiedot/

Liitteet

Liite 1. clairctl.groovy haavoittuvuusskannaus skripti

```
def vulnerabilitycheck(String CLAIRCTL_REGISTRY) {

PTCS_DOCKER_REGISTRY_ANALYSIS = CLAIRCTL_REGISTRY.replaceAll(/\//,'-')

FIXEDJOBNAME = JOB_NAME.replace(/\//,'/job/')


// Send Docker image layers to Clair and create HTML-report including vulnerabilities

sh """

clairctl --log-level Debug push --config /usr/share/clairctl.yml --local
${CLAIRCTL_REGISTRY}

clairctl --log-level Debug report --config /usr/share/clairctl.yml --format html --local
${CLAIRCTL_REGISTRY}:latest

"""


// HTML Publisher Plugin creates a link to Clairctl HTML-report

publishHTML (target: [

allowMissing: false,

alwaysLinkToLastBuild: true,

keepAll: true,

reportDir: 'reports/html',

reportFiles: 'analysis-*',

reportName: 'Vulnerability Report',

reportTitles: ''
```

```
])


// Unknown vulnerabilities
UNKNOWN_V = sh (
script: "grep -c '<div>Unknown</div>' reports/html/analysis-
${PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html | cat",
returnStdout: true
).trim()
echo "Unknown vulnerabilities: ${UNKNOWN_V}"


// Negligible vulnerabilities
NEGLIGIBLE_V = sh (
script: "grep -c '<div>Negligible</div>' reports/html/analysis-
${PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html | cat",
returnStdout: true
).trim()
echo "Negligible vulnerabilities: ${NEGLIGIBLE_V}"


// Low vulnerabilities
LOW_V = sh (
script: "grep -c '<div>Low</div>' reports/html/analysis-
${PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html | cat",
returnStdout: true
).trim()
echo "Low vulnerabilities: ${LOW_V}"
```

```
// Medium vulnerabilities

MEDIUM_V = sh (

script: "grep -c '<div>Medium</div>' reports/html/analysis-
${PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html | cat",

returnStdout: true

).trim()

echo "Medium vulnerabilities: ${MEDIUM_V}"


// High vulnerabilities

HIGH_V = sh (

script: "grep -c '<div>High</div>' reports/html/analysis-
${PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html | cat",

returnStdout: true

).trim()

echo "High vulnerabilities: ${HIGH_V}"


// Critical vulnerabilities

CRITICAL_V = sh (

script: "grep -c '<div>Critical</div>' reports/html/analysis-
${PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html | cat",

returnStdout: true

).trim()

echo "Critical vulnerabilities: ${CRITICAL_V}"
```

```
// Defcon1 vulnerabilities

DEFCON_V = sh (

script: "grep -c '<div>Defcon1</div>' reports/html/analysis-
${PTCS_DOCKER_REGISTRY_ANALYSIS}-latest.html | cat",

returnStdout: true

).trim()

echo "Defcon1 vulnerabilities: ${DEFCON_V}"


// Introduce previous environment variables to understandable format as integers.
Required for if/else to work.

int unknown_v = UNKNOWN_V as Integer

int negligible_v = NEGLIGIBLE_V as Integer

int low_v = LOW_V as Integer

int medium_v = MEDIUM_V as Integer

int high_v = HIGH_V as Integer

int critical_v = CRITICAL_V as Integer

int defcon_v = DEFCON_V as Integer


// Introduce vulnerability limits' environment variables to understandable format as
integers. Required for if/else to work.

int unknown_vulnerability_limit = UNKNOWN_VULNERABILITY_LIMIT as Integer

int negligible_vulnerability_limit = NEGLIGIBLE_VULNERABILITY_LIMIT as Integer

int low_vulnerability_limit = LOW_VULNERABILITY_LIMIT as Integer

int medium_vulnerability_limit = MEDIUM_VULNERABILITY_LIMIT as Integer

int high_vulnerability_limit = HIGH_VULNERABILITY_LIMIT as Integer
```

```
int critical_vulnerability_limit = CRITICAL_VULNERABILITY_LIMIT as Integer

int defcon_vulnerability_limit = DEFCON_VULNERABILITY_LIMIT as Integer


// Exit build if there are too many vulnerabilities of specific level

if (unknown_vulnerability_limit < unknown_v) {

echo 'Unknown: Too many vulnerabilities. Exiting build..'

sh 'exit 1'

}

else {

echo 'Unknown: OK.'

}


if (negligible_vulnerability_limit < negligible_v) {

echo 'Negligible: Too many vulnerabilities. Exiting build..'

sh 'exit 1'

}

else {

echo 'Negligible: OK.'

}


if (low_vulnerability_limit < low_v) {

echo 'Low: Too many vulnerabilities. Exiting build..'

sh 'exit 1'

}
```

```
else {
echo 'Low: OK.'
}

if (medium_vulnerability_limit < medium_v) {
echo 'Medium: Too many vulnerabilities. Exiting build..'
sh 'exit 1'
}
else {
echo 'Medium: OK.'
}

if (high_vulnerability_limit < high_v) {
echo 'High: Too many vulnerabilities. Exiting build..'
sh 'exit 1'
}
else {
echo 'High: OK.'
}

if (critical_vulnerability_limit < critical_v) {
echo 'Critical: Too many vulnerabilities. Exiting build..'
sh 'exit 1'
}
```

```
else {
echo 'Critical: OK.'
}

if (defcon_vulnerability_limit < defcon_v) {
echo 'Defcon1: Too many vulnerabilities. Exiting build..'
sh 'exit 1'
}
else {
echo 'Defcon1: OK.'
}

// Setting env for reporting
env.FIXEDJOBNAME = FIXEDJOBNAME
env.PTCS_DOCKER_REGISTRY_ANALYSIS = PTCS_DOCKER_REGISTRY_ANALYSIS
env.UNKNOWN_V = UNKNOWN_V
env.NEGLIGIBLE_V = NEGLIGIBLE_V
env.LOW_V = LOW_V
env.MEDIUM_V = MEDIUM_V
env.HIGH_V = HIGH_V
env.CRITICAL_V = CRITICAL_V
env.DEFCON_V = DEFCON_V
}
return this
```